Understanding Common Lisp

David J. Cooper, Jr.

August, 2000

Foreword[1]

Computers, and the software applications that power them, permeate every facet of our daily lives. From groceries to airline reservations to dental appointments, our reliance on technology is all-encompassing.

And, we want more. Every day, our expectations of technology and software increase:

- smaller cell phones that surf the net
- better search engines that generate information we actually want
- voice-activated laptops
- cars that know exactly where to go

The list is endless. Unfortunately, there is *not* an endless supply of programmers and developers to satisfy our insatiable appetites for new features and gadgets. "Cheap talent" to help complete the "grunt work" of an application no longer exists. Further, the days of unlimited funding are over. Investors want to see results, fast. Every day, hundreds of magazine and on-line articles focus on the time and people resources needed to support future technological expectations.

Common Lisp (CL) is one of the few languages and development options that can meet these challenges. Powerful, flexible, changeable on the fly — increasingly, CL is playing leading role in areas with complex problem-solving demands. Engineers in the fields of bioinformatics, scheduling, data mining, document management, B2B, and E-commerce have all turned to CL to complete their applications on time and within budget. But CL is no longer just appropriate for the most complex problems. Applications of modest complexity, but with demanding needs for fast development cycles and customization, are also ideal candidates for CL.

Other languages have tried to mimic CL, with limited success. Perl, Python, Java, C++, C# — they all incorporate some of the features that give Lisp its power, but their implementations tend to be brittle.

The purpose of this book is to showcase the features that make CL so much better than these imitators, and to give you a "quick-start" guide for using Common Lisp as a development environment. If you are an experienced programmer in languages other than Lisp, this guide gives you all the tools you need to begin writing Lisp applications. If you've had some exposure to Lisp in the past, this guide will help refresh those memories and shed some new light on CL for you.

But be careful, Lisp can be addicting! This is why many Fortune 500 companies will use nothing else on their 24/7, cutting-edge, mission-critical applications. After reading this book, trying our software, and experiencing a 3 to 10 times increase in productivity, we believe you will feel the same way.

[1] this Foreword was authored by Franz Inc.

Contents

Chapter 1

Introduction

1.1 The Past, Present, and Future of Common Lisp

1.1.1 Lisp Yesterday

Stanford University Professor John McCarthy discovered the basic principles of Lisp in 1958, when he was processing complex mathematical lists. Common Lisp (CL) is a high-level computer language, whose syntax follows a simple list-like structure. The term "Lisp" itself originally stood for "LISt Processing." When developing, testing, and running a CL program, at the core is a modern-day version of the original List Processor which processes (compiling, evaluating, etc.) the elements of your program. These elements, at the source code level, are represented as lists.A *list*, in this context, is just a sequence of items, much like a familiar shopping list or checklist. Originally the List was pretty much the only data structure supported by Lisp, but modern-day Common Lisp supports a wide range of flexible and efficient data structures.

In a sense, Lisp acts similarly to the concept of a Java Virtual Machine, but in a much more integral and natural fashion. The internal Lisp processor behaves like a complete operating system, with multiple threads of execution and the ability to load new code and redefine objects (functions, etc.) *dynamically*, i.e. without stopping and restarting the "machine."

This "operating system" characteristic also makes CL significantly more flexible than other popular programming languages. Whereas other OS languages, such as shell scripts or Perl CGI scripts, need to pipe data around, in CL all data manipulations can run interactively within one single process with a shared memory space.

1.1.2 Lisp Today

The most popular form of Lisp used today, "ANSI Common Lisp," was principally written and designed by Guy Steele in Common Lisp, the Language, 2nd Edition ("CLtL2") — (see the Bibliography in Appendix A). This version of Lisp became the accepted industry standard. In 1995, the American National Standards Institute recognized a slightly updated version as ANSI Common Lisp, the *first* object-oriented language to receive certification, and ANSI CL remains the *only* language that meets all of the criteria set forth by the Object Management Group (OMG) for a complete object-oriented language.

Paul Graham outlines ANSI CL in his 1996 book ANSI Common Lisp, also listed in Appendix A.

Official language standardization is important, because it protects developers from becoming saddled with legacy applications as new versions of a language are implemented. For example, this lack of standardization has been a continuing problem for Perl developers. Since each implementation of Perl defines the behavior of the language, it is not uncommon to have applications that

require outdated versions of Perl, making it nearly impossible to use in a mission-critical commercial environment.

1.1.3 Lisp Tomorrow

Many developers once hoped that the software development process of the future would be more automated through Computer-aided Software Engineering (CASE) tools. Such tools claim to enable programmers and non-programmers to diagram their applications visually and automatically generate code. While useful to a certain extent, traditional CASE tools cannot cover domain-specific details and support all possible kinds of customizations — so developers inevitably need to hand-code the rest of their applications, breaking the link between the CASE model and the code. CASE systems fall short of being a true "problem-solving tool."

The recursive nature of CL, and its natural ability to build applications in layers and build upon itself — in essence, *code which writes code which writes code...*, makes CL a much better software engineering solution. CL programs can generate other CL programs, allowing the developer to define *all* parts of the application in one unified high-level language. Using macros and functions, naturally supported by the CL syntax, the system can convert applications written in the high-level language automatically into "final" code. Thus, CL is both a programming language and a powerful CASE tool, and there is no need to break the connection.

Several other features of CL also save significant development time and manpower:

- Automatic Memory Management/Garbage Collection: inherent in CL's basic architecture.
- Dynamic Typing: In CL, values have types, but variables do not. This means that you don't have to declare variables, or placeholders, ahead of time to be of a particular type; you can simply create them or modify them on the fly[1].
- Dynamic Redefinition: New operations and functionality can be added to the running CL process without the need for downtime. In fact, a program can be running in one thread, while parts of the code are redefined in another thread. Moreover, as soon as the redefinitions are finished, the application will use the new code and is effectively modified while running. This feature is especially useful for server applications that can't afford downtimes – they can be patched and repaired while running.
- Portability: The "virtual machine" aspect of CL makes CL programs especially portable.
- Standard Features: CL contains many built-in features and data structures that are non-standard in other languages. Features such as full-blown symbol tables and package systems, hash tables, list structures, and a comprehensive object system are several "automatic" features which would require significant developer time to recreate in other languages.

1.2 Convergence of Hardware and Software

The most common criticism of Lisp usually made by programmers unfamiliar with the language is that Lisp applications are too big and too slow. While this may have been true 20 years ago, it is *absolutely* not the case today. Today's inexpensive commodity computers have more memory than some of the most powerful machines available just five years ago, and easily meet the computing needs of CL. Hardware is no longer the critical factor, programmers are!

Further, CL programming teams tend to be small, because the inherent features in CL empower developers to do more. Without excessive effort, a single CL developer can create and deploy a sophisticated, powerful program in a much shorter period of time than if another language were used.

[1] While you don't *have* to declare variable types, you *may* declare them. Doing so can help CL's compiler to optimize your program further.

1.3 The CL Model of Computation

Programming in CL is distinguished from programming in other languages due to its unique syntax and development mode. CL allows developers to make changes and test them *immediately*, in an incremental manner. In other languages, a developer must follow the "compile-link-run-test" cycle. These extra steps add minutes or hours to *each cycle* of development, and break the programmer's thought process. Multiply these figures by days, weeks, and months — and the potential savings are phenomenal.

Chapter 2

Operating a CL Development Environment

This chapter covers some of the hands-on techniques used when working with a CL development environment, specifically, the Allegro CL® environment from Franz Inc. If you are completely unfamiliar with any Lisp language, you may wish to reference Chapter 3 as you read this chapter.

Ideally, while reading this chapter, you should have access to an actual Common Lisp session and try typing in some of the examples.

When working with Common Lisp, it helps to think of the environment as its own operating system, sitting on top of whatever operating system you happen to be working with. In fact, people used to build entire *workstations* and *operating systems* in Lisp. The Symbolics Common Lisp environment still runs as an emulated machine on top of the 64-bit Compaq/DEC Alpha CPU, with its own Lisp-based operating system.

A complete and current listing of available CL systems can be found on the Association of Lisp Users website, at

`http://www.alu.org/table/systems.htm`

These systems follow the same basic principles, and most of what you learn and write on one system can be applied and ported over to others. The examples in this guide were prepared using Allegro CL for Linux. Where possible, we will note Allegro CL-specific syntax and extensions.

2.1 Installing a CL Environment

Installing a CL environment is generally a straightforward process. For Allegro CL on Linux, this basically involves running a simple shell script which steps you through some licensing information, then proceeds to unpack the distribution, and sets up the CL executable and supporting files.

The Windows installation involves somewhat less typing and more mouse clicking.

2.2 Running CL in a Shell Window

The most primitive and simplest way to run CL is directly from the Unix (Linux) shell from within a terminal window. Most programmers do not work this way on a daily basis because other modes of use provide more power and convenience with fewer keystrokes.

This mode of working is sometimes useful for logging into a running CL system on a remote host (e.g. a CL process acting as a webserver) when using a slow dial-up connection.

2.2.1 Starting CL from a Terminal Window

To start a CL session from a terminal window, simply type

```
lisp
```

from the Unix (or DOS) command line. Assuming your shell execution path is set up properly and CL is installed properly, you will see some introductory information, then will be presented with a Command Prompt which should look very similar to the following:

```
USER(1):
```

CL has entered its *read-eval-print* loop, and is waiting for you to type something which it will then *read*, *evaluate* to obtain a *return-value*, and finally *print* this resulting return-value.

The `USER` printed before the prompt refers to the current CL *package* (more on Packages later), and the number (`1`) simply keeps track of how many commands have been entered at the prompt.

2.2.2 Stopping CL from a Terminal Window

Now that you have learned how to start CL, you should probably know how to stop it. In Allegro CL, one easy way to stop the CL process is to type

```
(excl:exit)
```

at the Command Prompt. This should return you to the shell. In very rare cases, a stronger hammer is required to stop the CL process, in which case you can try typing

```
(excl:exit 0 :no-unwind t)
```

Try It: Now try starting and stopping CL several times, to get a feel for it. In general, you should notice that CL starts much more quickly on subsequent invocations. This is because the entire executable file has already been read into the computer's memory and does not have to be read from the disk every time.

A shortcut for the `(excl:exit)` command is the *toplevel* command `:exit`. Toplevel commands are words which are preceded by a colon [:] that you can type at the CL Command Prompt as a shorthand way of making CL do something.

Try It: Start the CL environment. Then, using your favorite text editor, create a file `/tmp/hello.lisp` and place the following text into it (don't worry about understanding the text for now):

```
(in-package :user)

(defun hello ()
  (write-string "Hello, World!"))
```

Save the file to disk. Now, at the CL prompt in the shell where you started the CL environment, compile and load this file as follows *(text after the "`USER`" prompt represents what you have to type; everything else is text printed by CL):*

```
USER(3): (compile-file "/tmp/hello")
;;; Compiling file /tmp/hello.lisp
;;; Writing fasl file /tmp/hello.fasl Fasl write complete
#p"/tmp/hello.fasl"
NIL
NIL

USER(4): (load "/tmp/hello")
; Fast loading /tmp/hello.fasl
T
```

By default, the `compile-file` command will look for files with the `.lisp` suffix, and the `load` command will load the resulting compiled machine-code (binary) file, which by default will have the extension `.fasl`. `fasl` stands for "FASt Loading" file.

Note that, in general, simply loading an uncompiled Lisp file (using `load`) will functionally have the same effect as loading a compiled binary (`fasl`) file. But the `fasl` file will load faster, and any Functions, Objects, etc. defined in it will perform faster, since they will have been optimized and translated into language which is closer to the actual machine. Unlike Java "`class`" files, CL `fasl` files usually represent native machine-specific code. This means that compiled CL programs will generally run much faster than compiled Java programs, but CL programs must be compiled separately for each type of machine on which you want to run them.

Finally, try executing the newly defined Function `hello` by typing the following at your command line:

```
USER(5): (hello)
Hello, World!
"Hello, World!"
```

You should see the String `Hello, World!` printed (without double-quotes), then the returned String (with double-quotes) will be printed as well. In the next chapter, The CL Language, we will learn more about exactly what is going on here. For now the main point to understand is the idea of compiling and loading definitions from files, and invoking Functions by typing expressions at the CL command line.

2.3 Running CL inside a Text Editor

A more powerful way of developing with CL is by running it from within a Unix (or DOS) shell that is running inside a buffer in a powerful text editor, such as Gnu Emacs[1]. One advantage of this approach is that the editor (in our example, Emacs) keeps a history of the entire session: both the expressions the programmer has typed, as well as everything that CL has printed. All these items can be easily reviewed or recalled.

2.3.1 A Note on Emacs and Text Editors

To develop using Common Lisp, you can, of course, use any text editor of your choice, although Emacs happens to be especially synergistic for working with CL, for reasons we will discuss later.

[1] Gnu Emacs comes pre-installed with all Linux systems, and for other systems should come as part of your Allegro CL distribution. In any case, Emacs distributions and documentation are available from `http://www.gnu.org`

With a bit of practice, you will find that the combination of Emacs and Allegro CL provides a more powerful and convenient development environment than any of today's flashy "visual programming" environments. If you are used to working with the "vi" editor, Emacs comes with several "vi" emulation modes which you may want to experiment with on a transitional basis. If you are used to working with a standard Windows-based text editor, you will find a smooth transition to Emacs, and will find many "surprise and delight" features as you progress on your Emacs journey.

This guide provides examples using Gnu Emacs.

To work with CL in an Emacs shell buffer, you should be reasonably familiar with the text editor. If you are completely new to Gnu Emacs, go through the Emacs Tutorial, available under the "Help" menu at the top of the Emacs window.

2.3.2 Emacs Terminology

As you will learn in the Emacs Tutorial, when working with Emacs you make frequent use of *key chords*. Similar to "chords" on a piano, keychords can allow some powerful text processing with relatively few sequential keystrokes.

They are key combinations involving holding down the `Control` key and/or the `Meta` key, and pressing some other regular key.

See Appendix B for information on which keys represent `Control` and `Meta` on your keyboard, and how you can customize these. To begin, try using the key labeled "Control" for `Control` and the keys labeled "Alt" (or the diamond-shaped key on Sun keyboards) for `Meta`. If your keyboard has a "Caps Lock" key to the left of the letter "A," you should consider remapping this to function as `Control`, as explained in Appendix B.

This guide uses the following notations to indicate keychords (identical to those referenced in the Emacs Tutorial):

- `M-x` means to hold down the `Meta` key, and press (in this case) the "X" key
- `C-x` means to hold down the `Control` key, and press (in this case) the "X" key
- `C-M-q` means to hold down the `Control` key *and* the `Meta` key at the *same time* , and press (in this case) the "Q" key
- `M-A` would mean to hold down the `Meta` key, *and* the `Shift` key (because the "A" is uppercase), and press (in this case) the "A" key.

2.3.3 Starting, Stopping, and Working With CL inside an Emacs Shell

To start Emacs, type

```
emacs
```

or

```
gnuemacs
```

then start a Unix (or DOS) shell within emacs by typing `M-x shell`. Now type

```
lisp
```

inside this shell's window to start a CL session there. To shut down a session, exit CL exactly as above by typing

```
(excl:exit)
```

Exit from the shell by typing

```
exit
```

and finally exit from Emacs by typing `C-x C-c` or `M-x kill-emacs`. Note that it is good practice always to exit from CL before exiting from Emacs. Otherwise the CL process may be left as an "undead" (zombie) process.

When in the CL process, you can move forward and backward in the "stack" (history) of expressions that you have typed, effectively recalling previous commands with only a few keystrokes. Use `M-p` to move backward in the history, and `M-n` to move forward. You can also use all of the editor's text processing commands, for example to cut, copy, and paste expressions between the command line and other windows and open files ("buffers") you may have open.

Try It: try typing the following expressions at the CL prompt. See what return-values are printed, and try recalling previous expressions (commands) using `M-p` and `M-n`:

```
(list 1 2 3)

(+ 1 2 3)

(> 3 4)

(< 3 4)
```

Now, of course, you can edit, compile, load, and test your `hello.lisp` file as we did in Section 2.2. But this time you are staying within a single environment (the text editor environment) to achieve all these tasks.

2.4 Running CL as a subprocess of Emacs

An even more powerful way of developing with Allegro CL is by running it within Emacs in conjunction with a special Emacs/Lisp interface provided by Franz Inc. This technique provides all the benefits of running in an Emacs shell as outlined above, but is more powerful because the Emacs-CL interface extends Emacs with several special commands. These commands interact with the CL process, making the editor appear to be "Lisp-aware." The combination of Emacs with the Emacs-CL interface results in a development environment whose utility approaches that of the vintage Symbolics Lisp Machines, which many Lisp aficionados still consider to be technology advanced well beyond anything produced today.

Complete documentation of the Emacs-CL interface can be viewed by pointing your web browser at: `file:/usr/local/acl5/eli/readme.htm` (you may have to modify this pathname to point to your actual installed location of Allegro CL (`acl5`)).

2.4.1 Starting the CL subprocess within Emacs

To enable your Emacs to function in the proper manner, place the following expressions in a file named `.emacs` in your home directory:

Keychord	*Action*	*Emacs-Lisp Function*
`C-M-x`	Compiles and loads the Function near the Emacs point	`fi:lisp-eval-or-compile-defun`
`C-c C-b`	Compiles and loads the current Emacs buffer	`fi:lisp-eval-or-compile-current-buffer`
`M-A`	Shows the Argument List for a CL Function	`fi:lisp-arglist`
`C-c .`	Finds the definition of a CL object	`fi:lisp-find-definition`

Table 2.1: Common Commands of the Emacs-Lisp Interface

```
(setq load-path
  (cons "/usr/local/acl5/eli" load-path))
(load "fi-site-init")
```

As above, you may have to modify this pathname to point to your actual installed location of Allegro CL (`acl5`). Once you have added these lines to your `.emacs` file, you should be able to restart emacs, then issue the command:

```
M-x fi:common-lisp
```

to start CL running and automatically establish a network connection between the Emacs process and the CL process (in theory, the two processes could be running on different machines, but in practice they usually will run on the same machine). Accept all the defaults when prompted.

You can also start emacs with the CL subprocess automatically, by invoking emacs with an optional "function" argument, as follows:

```
emacs -f fi:common-lisp
```

This will start Emacs, which itself will immediately launch CL.

2.4.2 Working with CL as an Emacs subprocess

Once you have started CL within Emacs, you will be presented with a special buffer named `*common-lisp*` which in many ways is similar to simply running CL within a shell within Emacs as above. But now you have several more capabilities available to you, as outlined in Table 2.1.

To summarize, the Lisp-Emacs interface adds capabilities such as the following to your development environment:

- Start and stop the CL process using Emacs commands
- Compile and load files into the running CL process, directly from the open Emacs buffer you are editing (no need explicitly to save the file, call the `compile-file` Function, then call the `load` Function, as with the "in-a-shell" techniques above)
- Query the running CL process for information about objects currently defined in it — for example, querying the CL, directly from Emacs, for the argument List of a particular Function

- Run multiple *read-eval-print* loops, or *listeners*, in different threads of CL execution, as Emacs buffers
- Compile and load specified sections of an open Emacs buffer
- Locate the source code file corresponding to any defined CL object, such as a Function or parameter, and automatically open it in an Emacs buffer
- Perform various debugging actions and other general CL help capabilities, directly from within the Emacs session

In short, Emacs becomes a remarkably powerful, infinitely customizable, CL integrated development environment.

2.4.3 Compiling and Loading a File from an Emacs buffer

Try it: Try starting Emacs with a CL subprocess as outlined above. If you have trouble, consult the URL listed above for complete instructions on running the Lisp-Emacs interface.

Once you have Emacs running with a CL subprocess, you should have a buffer named `*common-lisp*`. If your Emacs session is not already visiting this buffer, visit it with the command `C-x b *common-lisp*`, or select `*common-lisp*` from the "Buffers" menu in Emacs (see Appendix B for instructions on how to set up a "hot key" for this and other common tasks when working with the Emacs-Lisp interface).

Now, split your Emacs frame into two windows, with the command `C-x 2`. In both windows you should now see `*common-lisp*`. Now move the point[2] to the other window with `C-x o`, and open the `hello.lisp` file you created for the previous exercise[3]. If you don't have that file handy, create it now, with the following contents:

```
(defun hello ()
  (write-string "Hello, World!"))
```

Now, compile and load the contents of this Function definition with `C-M-x`. Finally, switch to the `*common-lisp*` window again with `C-x o`, and try invoking the `hello` Function by calling it in normal Lisp fashion:

```
USER(5): (hello)
Hello, World!
"Hello, World!"
```

You should see the results printed into the same `*common-lisp*` buffer.

Try it: Now try the following: go back to the `hello.lisp` buffer, make an edit to the contents (for example, change the String to `"Hello, CL World!"`). Now compile the buffer with this change (Note that you do not necessarily have to save the file). Finally, return to the `*common-lisp*` buffer, and try invoking the Function again to confirm that the redefinition has taken effect.

You have now learned the basic essentials for working with the Emacs-Lisp interface. See Appendix B for more information about Emacs customization and convenient Emacs commands specific to the Lisp editing mode.

[2] In technical Emacs parlance, the *point* is the insertion point for text, and the *cursor* is the mouse pointer.

[3] Remember that pressing the space bar will automatically complete filenames for you when opening files in Emacs.

2.5 Integrated Development Environment

Some CL implementations support or integrate with an *IDE*, or *Integrated Development Environment*. Such environments provide visual layout and design capabilities for User Interfaces and other graphical, visual techniques for application development. For example, such a system is available for Allegro CL on the Microsoft platforms. The details of these systems are covered in their own documentation and are beyond the scope of this guide.

2.6 The User Init File

When CL begins execution, it usually looks for an *initialization file*, or "init" file, to load. When CL loads this file, it will read and evaluate all the commands contained in it.

The default name and location for this initialization file is

```
.clinit.cl
```

in the user's home directory on Unix/Linux, and

```
c:\clinit.cl
```

on Windows systems.

For example, the init file can be used to define library Functions and variables that the user employs frequently, or to control other aspects of the CL environment. An example init file is:

```
(in-package :user)

(defun load-project ()
  (load "~/lisp/part1")
  (load "~/lisp/part2"))
```

This will have the effect of defining a convenient Function the user can then invoke to load the files for the current project. Note the call to the `in-package` Function, which tells CL which Package to make "current" when loading this file. As we will see in more detail in Section 3.7, CL Packages allow programs to be separated into different "layers," or "modules." Notice also that in the calls to `load` we do not specify a file "type" extension, e.g. `.lisp` or `.fasl`. We are making use of the fact that the `load` Function will look first for a binary file (of type `fasl`), and if this is not found it will then look for a file of type `lisp`. In this example we assume that our Lisp files will already have been compiled into up-to-date binary `fasl` files.

In Section 2.9 we will look at some techniques for managing larger projects, and automating both the compiling and loading process for the source files making up a project.

Note: the IDE mentioned in Section 2.5 is also a full-featured and powerful project system. The IDE is described in its own documentation, and is beyond the scope of this document.

2.7 Using CL as a scripting language

Although CL is usually used in the interactive mode described above while developing a program, it can be also used as a scripting language in more of a "batch" mode. For example, by running CL as:

```
lisp -e '(load "~/script")' < datafile1 > datafile2
```

it will load all of the Function definitions and other expressions in file `script.fasl` or `script.lisp`, will process `datafile1` on its Standard Input, and will create `datafile2` with anything it writes to its Standard Output. Here is an example of a simple `script.lisp` file:

```
(in-package :user)

(defun process-data ()
  (let (x j)
    ;;Read a line of input from datafile1
    (setq x (read-line))
    ;;Remove leading spaces from X
    (setq j (dotimes (i (length x))
              (when (not (string-equal " " (subseq x i (+ i 1))))
                (return i))))
    (setq x (subseq x j))
    ;;Write the result to datafile2
    (format t "~A~%" x)))

 (process-data)
```

This file defines a Function to process the datafile, then calls this Function to cause the actual processing to occur.

Note the use of the semicolon (";"), which is a *reader macro* which tells CL to treat everything between it and the end of the line as a comment (to be ignored by the CL Reader).

2.8 Debugging in CL

CL provides one of the most advanced debugging and error-handling capabilities of any language or environment. The most obvious consequence of this is that your application will *very* rarely "crash" in a fatal manner. When an error occurs in a CL program, a *break* occurs. CL prints an error message, enters the debugger, and presents the user with one or more possible restart actions. This is similar to running inside a debugger, or in "debug mode," in other languages, but in CL this ability comes as a built-in part of the language.

For example, if we call the `+` Function with an alphabetic Symbol instead of a Number, we generate the following error:

```
USER(95): (+ 'r 4)
Error: `R' is not of the expected type `NUMBER'
   [condition type: TYPE-ERROR]

Restart actions (select using :continue):
  0: Return to Top Level (an "abort" restart)
[1] USER(96):
```

The number `[1]` at the front of the prompt indicates that we are in the debugger, at Level 1. The debugger is itself a CL Command Prompt just like the toplevel Command Prompt, but it has some

additional functionality. The next section provides a partial list of commands available to be entered directly at the debugger's Command Prompt:

2.8.1 Common debugger commands

- *:reset* returns to the toplevel (out of any debug levels)
- *:pop* returns up one level
- *:continue* continues execution after a break or an error
- *:zoom* Views the current Function call stack[4]
- *:up* Moves the currency pointer up one frame in the stack
- *:down* Moves the currency pointer down one frame in the stack
- *:help* Describes all debugger commands

Franz Inc.'s Allegro Composer product also offers graphical access to these commands, as well as additional functionality.

2.8.2 Interpreted vs Compiled Code

CL works with both *compiled* and *interpreted* code. When you type expressions at a Command Prompt, or use CL's `load` Function to load `lisp` source files directly, you are introducing interpreted code into the system. This code does not undergo any kind of optimization or translation into a lower-level machine representation — internally to CL it exists in a form very similar to its source code form. CL must do a lot of work every time such code is evaluated, since it must be translated into lower-level machine instructions each time.

When you create compiled code, on the other hand, for example by using CL's `compile-file` command in order to produce a `fasl` file, the code is optimized and translated into an efficient form. Loading the `fasl` file will replace any previous interpreted definitions with compiled definitions.

Interpreted code is the best to use for debugging a program, because the code has not gone through as many transformations, so the debugger can provide the programmer with more recognizable information.

If you are debugging with compiled code (e.g. after loading a `fasl` file), and you feel that not enough information is available from within the debugger, try redefining the Function in interpreted form and running it again. To redefine a Function in interpreted form, of course, you can simply `load` the `lisp` source file, rather than the compiled `fasl` file.

2.8.3 Use of (break) and C-c to interrupt

In order to cause a break in your program interactively, issue the `C-c` command. When running inside Emacs, you must use a "double" `C-c`, because the first `C-c` will be intercepted by Emacs. Depending on what your application is doing, it may take a few moments for it to respond to your break command. Once it responds, you will be given a normal debugger level, as described above.

2.8.4 Profiling

CL has built-in functionality for monitoring its own performance. The most basic and commonly used component is the `time` Macro. You can "wrap" the `time` Macro around any expression. Functionally, `time` will not affect the behavior of the expression, but it will print out some information about how long the expression took to be evaluated:

[4] the *call stack* is the sequence of Functions calls leading up to the error or break.

```
USER(2): (time (dotimes (n 10000) (* n 2)))
; cpu time (non-gc) 130 msec user, 20 msec system
; cpu time (gc)     0 msec user, 0 msec system
; cpu time (total)  130 msec user, 20 msec system
; real time  147 msec
```

It is often interesting to look at the difference in `time` between interpreted and compiled code. The example above is interpreted, since we typed it directly at the Command Line, so CL had no opportunity to optimize the `dotimes`. In compiled form, the above code consumes only one millisecond instead of 147 milliseconds.

In addition to the basic `time` Macro, Allegro CL provides a complete Profiler package, which allows you to monitor exactly what your program is doing, when it is doing it, how much time it takes, and how much memory it uses.

2.9 Developing Programs and Applications in CL

For developing a serious application in CL, you will want to have a convenient procedure in place for starting up your *development session*.

2.9.1 A Layered Approach

The development session consists of your favorite text editor with appropriate customizations, and a running CL process with any standard code loaded into it. While you can certainly develop serious applications on top of base CL, typically you will work in a *layered* fashion, on top of some standard tools or libraries, such as an embedded language, a webserver, etc.

Using one of the techniques outlined below, starting a customized development session with your standard desired tools should become convenient and transparent.

2.9.2 Compiling and Loading your Project

An individual project will generally consist of a *tree* of directories and files in your computer's filesystem. We will refer to this directory tree as the project's *codebase*.

Often, it does not matter in what *order* the files in the codebase are compiled and loaded. Typical CL definitions, such as Functions, Classes, etc., can often be defined in any order. However, certain constructs, most notably Packages and Macros, *do* have order dependencies. In general, CL Packages and Macros must be defined before they can be used, or referenced.

So one of the requirements of the startup process for a development session is that all files be compiled and/or loaded in the correct order.

To start a development session on your project, you will generally want CL to do the following:

1. Go through your project's Codebase, finding source files in the proper order
2. For each file found, either load the corresponding binary file (if it is up-to-date), or compile the source file to create a new binary file and load it (if the source file has changed since the most recent compilation)

While CL does not dictate one standard way of accomplishing this task, several options are available. One is to use a *defsystem* package, which will allow you to prepare a special file, similar to a "make" file, which lists out all your project's source files and their required load order. Allegro CL contains its own Defsystem package as an extension to CL, and the open-source MK:Defsystem is available through Sourceforge (`http://www.sourceforge.net`).

Using a Defsystem package requires you to maintain a catalog listing of all your project's source files. Especially in the early stages of a project, when you are adding, renaming, and deleting a lot of files and directories, maintaining this catalog listing by hand can become tiresome.

For this purpose, you may wish to use a lighter-weight utility for compiling and loading your files, at least in the early stages of development. For very small projects, you can use the Allegro CL Function `excl:compile-file-if-needed`, an extension to CL, by writing a simple Function which calls this Function repeatedly to compile and load your project's files.

The Bootstrap package, available at `http://gdl.sourceforge.net`, provides the Function `cl-lite`, which will traverse a codebase, compiling and loading files, and observing simple "ordering directive" files which can be placed throughout the codebase to enforce correct load ordering.

For a new project, the best approach is probably to start with a simple but somewhat manual loading technique, then graduate into more automated techniques as they become necessary and as you become more familiar with the environment.

2.9.3 Creating an Application "Fasl" File

`Fasl` files in Allegro CL and most other CL systems have the convenient property that they can simply be concatenated together, e.g. with the Unix/Linux `cat` command, to produce a *single* `fasl` file. Loading this single file will have the same effect as loading all the individual files which went into it, in the same order as they were concatenated.

In certain situations, this can be a convenient mechanism for loading and/or distributing your project. For example, you can prepare a concatenated `fasl` file, email it to a colleague, and the colleague needs only to load this file. There is no chance of confusion about how (e.g. in what order) to load individual files.

2.9.4 Creating an Image File

Loading Lisp source files or compiled binary files into memory at the start of each session can become slow and awkward for large programs.

An alternative approach is to build an *image file*. This entails starting a CL process, loading all application files into the running CL process, then creating a new "CL Image." In Allegro CL, this is done by invoking the Function:

```
excl:dumplisp
```

The Image File is typically named with a `.dxl` extension. Once you have created an Image File, you can start a session simply by starting CL with that Image File, as follows:

```
lisp -I <image-file-name>.dxl
```

You must then compile/load any files that have changed since the Image File was created. Starting CL with the Image File has the same effect as starting a base CL and loading all your application files, but is simpler and much faster.

2.9.5 Building Runtime Images

CL Image files built with `excl:dumplisp` contain the entire CL environment, and may also contain information which is specific to a certain machine or certain site. Allegro CL also provides a mechanism for building *runtime images*, which contain only the information needed to run an end-user application. These Runtime Images can be packaged and distributed just as with any stand-alone software application.

Preparing runtime images for different platforms (e.g. Sun Solaris, Linux, Windows, etc.) is usually a simple matter of running a compilation and building a runtime image for the target platform — application source code typically remains identical among different platforms. See your platform-specific documentation for details about creating Runtime Images.

2.9.6 Using an Application Init File

In some cases, for a finished production application, you will want the CL process to execute certain commands upon startup (for example, loading some data from a database). However, you do not want to depend on the end-user having the appropriate `.clinit.cl` in his or her home directory. For such cases, you can specify a different location for the init file, which can be installed along with the application itself. One way to accomplish this is with the `-q` command-line argument to the `lisp` command. This argument will tell CL to look in the *current* directory, rather than in the default location, for the init file.

To use this technique, you would set up an "application home" directory, place the application's `.clinit.cl` file in this directory, and start the application with a production startup script. This script would first change into the "application home" directory, then start CL with your application's image file, or start a Runtime Image. Such a startup script might look like this:

```
cd $MYAPP_HOME
lisp -I image.dxl -q
```

In this example, both the application's init file and the application's image file would be housed in the directory named by the environment variable `$MYAPP_HOME`.

2.10 Using CL with Other Languages and Environments

One of CL's strong points is its flexibility in working with other environments. In this section we will briefly mention six different ways in which CL can integrate and/or communicate with other languages and environments.

2.10.1 Interfacing with the Operating System

One of the most basic ways to interact with the outside world is simply to invoke commands at the operating system level, in similar fashion to Perl's "system" command. In Allegro CL, one way to accomplish this is with the Function `excl:run-shell-command`. This command takes a String which represents a command (possibly with arguments) to be executed through an operating system shell, such as the "C" shell on Unix:

```
(excl:run-shell-command "ls")
```

The Standard Output from a shell command invoked in this fashion will be inherited by CL's Standard Output, so you will see it in the normal CL console. Alternatively, you can specify that the output goes to another location, such as a file, or a variable defined inside the CL session.

When using `excl:run-shell-command`, you can also specify whether CL should wait for the shell command to complete, or should simply go about its business and let the shell command complete asynchronously.

2.10.2 Foreign Function Interface

Allegro CL's *foreign function interface* allows you to load compiled libraries and object code from C, C++, Fortran, and other languages, and call functions defined in this code as if they were defined natively in CL. This technique requires a bit more setup work than using simple shell commands with `excl:run-shell-command`, but it will provide better performance and more transparent operation once it is set up.

2.10.3 Interfacing with Corba

Corba, or the Common Object Request Broker Architecture, is an industry-standard mechanism for connecting applications written in different object-oriented languages.

To use Corba, you define a standard Interface for your application in a language called Interface Definition Language (Corba IDL), and each application must compile the standard Interface and implement either a Servant or a Client for the interface. In this manner, Servants and Clients can communicate through a common protocol, regardless of what langauge they are written in.

In order to use Corba with CL requires a Corba IDL compiler implemented in CL, as well as a run-time ORB, or Object Request Broker, to handle the actual Client/Server communications. Several such Corba IDL compilers and ORBs are available for CL. An example of such a system is Orblink, which is available as an optional package for Allegro CL. Orblink consists of a complete Corba IDL compiler, and a CL-based ORB, which runs as a CL thread rather than as a separate operating system process.

Using Orblink to integrate a CL application to a Corba Client/Server environment is a straightforward process. To start, you simply obtain the Corba IDL files for the desired Interface, and compile them in CL with the command `corba:idl`. This will automatically define a set of Classes in CL corresponding to all the Classes, Methods, etc. which make up the Interface.

To use CL as a Client, you can now simply make instances of the desired Classes, and use these to call the desired Methods. The Orblink ORB will take care of communicating these Method calls across the network, where they will be invoked in the actual process where the corresponding Servant is implemented (which could be on a completely different machine written in a completely different language).

To use CL as a Servant, you must implement the relevant Interfaces. This consists of defining Classes in CL which *inherit* from the "stub" Servant Classes automatically defined by the compilation of the Corba IDL, then defining Methods which operate on these Classes, and which perform as advertised in the Interface. Typically, a Corba Interface will only *expose* a small piece of an entire application, so implementing the Servant Classes would usually represent a small chore, relative to the application itself.

2.10.4 Custom Socket Connections

Just about every CL implementation provides a mechanism to program directly with network sockets, in order to implement "listeners" and network client applications.

As a practical example, Figure 2.1 shows a Telnet server, written by John Foderaro of Franz Inc., in 22 lines of code in Allegro CL.

This Telnet server will allow a running CL process to accept a Telnet login on port 4000 on the machine on which it is running, and will present the client with a normal CL Command Prompt (for security, it will by default only accept connections from the local machine).

2.10.5 Interfacing with Windows (COM, DLL, DDE)

Most Commercial CL implementations for the Microsoft Windows platform will allow CL applications to use various Microsoft-specific means for integrating with the Microsoft platform. For

```
(defun start-telnet (&optional (port 4000))
  (let ((passive (socket:make-socket :connect :passive
                                     :local-host "127.1"
                                     :local-port port
                                     :reuse-address t)))
    (mp:process-run-function
     "telnet-listener"
     #'(lambda (pass)
         (let ((count 0))
           (loop
             (let ((con (socket:accept-connection pass)))
               (mp:process-run-function
                (format nil "tel~d" (incf count))
                #'(lambda (con)
                    (unwind-protect
                        (tpl::start-interactive-top-level
                         con
                         #'tpl::top-level-read-eval-print-loop
                         nil)
                      (ignore-errors (close con :abort t))))
                con)))))
     passive)))
```

Figure 2.1: Telnet Server in 22 Lines

example, an Allegro CL application can be set up to run as a COM Server, or compiled into a DLL to be used by other applications.

2.10.6 Code Generation into Other Languages

CL excels at reading and generating CL code. It is also a powerful tool for analyzing and generating code in other languages.

An example of this can be found in Chapter 5, where we use a convenient CL macro to generate HTML for a web page.

Another example is MSC's AutoSim product, which uses a CL program to model an automobile, then generates optimized C programs for solving specialized systems of linear equations relating to vehicle dynamics.

CL also played a crucial role in preventing the Y2K disaster by automatically analyzing and repairing millions of lines of COBOL!

Chapter 3

The CL Language

If you are new to the CL language, we recommend that you supplement this chapter with other resources. See the Bibliography in Appendix A for some suggestions. The Bibliography also lists two interactive online CL tutorials from Tulane University and Texas A&M, which you may wish to visit.

In the meantime, this chapter will provide a condensed overview of the language. Keep in perspective, however, that this booklet is intended as a summary, and will not be able to go into some of the more subtle and powerful techniques possible with Common Lisp.

3.1 Overview of CL and its Syntax

The first thing you should observe about CL (and most languages in the Lisp family) is that it uses a generalized *prefix* notation.

One of the most frequent actions in a CL program, or at the toplevel *read-eval-print* loop, is to call a *Function*. This is most often done by writing an *expression* which names the Function, followed by its arguments. Here is an example:

```
(+ 2 2)
```

This expression consists of the Function named by the Symbol "`+`," followed by the arguments `2` and another `2`. As you may have guessed, when this expression is evaluated it will return the value `4`.

Try it: Try typing this expression at your command prompt, and see the return-value being printed on the console.

What is it that is actually happening here? When CL is asked to *evaluate* an *expression* (as in the toplevel *read-eval-print* loop), it evaluates the expression according to the following rules:

1. If the expression is a *number* (i.e. looks like a Number), it simply evaluates to itself (a Number):

   ```
   USER(8): 99
   99
   ```

2. If the expression looks like a *string* (i.e. is surrounded by double-quotes), it also simply evaluates to itself:

```
USER(9): "Be braver -- you can't cross a chasm in two small jumps."
"Be braver -- you can't cross a chasm in two small jumps."
```

3. If the expression looks like a literal *symbol*, it will simply evaluate to that *symbol* (more on this in Section 3.1.4):

```
USER(12): 'my-symbol
MY-SYMBOL
```

4. If the expression looks like a *list* (i.e. is surrounded by parentheses), CL assumes that the *first* element in this List is a *symbol* which names a *Function* or a *Macro*, and the *rest* of the elements in the List represent the *arguments* to the Function or Macro. (We will discuss *Functions* first, *Macros* later). A *Function* can take zero or more arguments, and can *return* zero or more *return-values*. Often a Function only returns one return-value:

```
USER(14): (expt 2 5)
32
```

Try it: Try typing the following functional expressions at your command prompt, and convince yourself that the printed return-values make sense:

```
(+ 2 2)

(+ 2)

2

(+)

(+ (+ 2 2) (+ 3 3))

(+ (+ 2 2))

(user-id)

(user-homedir-pathname)

(get-universal-time) ;; returns number of seconds since January 1, 1900

(search "Dr." "I am Dr. Strangelove")

"I am Dr. Strangelove"

(subseq (search "Dr." "I am Dr. Strangelove")
        "I am Dr. Strangelove")
```

3.1.1 Evaluation of Arguments to a Function

Note that the arguments to a Function can *themselves* be any kind of the above expressions. They are evaluated in order, from left to right, and finally they are passed to the Function for it to evaluate. This kind of nesting can be arbitrarily deep. Do not be concerned about getting lost in an ocean of parentheses — most serious text editors can handle deeply nested parentheses with ease, and moreover will automatically indent your expressions so that you, as the programmer, never have to concern yourself with matching parentheses.

3.1.2 Lisp Syntax Simplicity

One of the nicest things about Lisp is the simplicity and consistency of its syntax. What we have covered so far about the evaluation of arguments to Functions is *just about everything you need to know* about the syntax. The rules for Macros are very similar, with the primary difference being that all the arguments of a Macro are not necessarily evaluated at the time the Macro is called — they can be transformed into something else first.

This simple consistent syntax is a welcome relief from other widely used languages such as C, C++, Java, Perl, and Python. These languages claim to use a "more natural" "infix" syntax, but in reality their syntax is a confused mixture of prefix, infix, and postfix. They use infix only for the simplest arithmetic operations such as

```
2 + 2
```

and

```
3 * 13
```

and use postfix in certain cases such as

```
i++
```

and

```
char*
```

But mostly they actually use a prefix syntax much the same as Lisp, as in:

```
split(@array, ":");
```

So, if you have been programming in these other languages, you have been using prefix notation all along, but may not have noticed it! If you look at it this way, the prefix notation of Lisp will seem much less alien and, in fact, downright natural to you.

3.1.3 Turning Off Evaluation

Sometimes you want to specify an expression at the toplevel *read-eval-print* loop, or inside a program, but you do not want CL to evaluate this expression. You want just the literal expression. CL provides the special operator `quote` for this purpose. For example,

```
(quote a)
```

will return the literal Symbol `A`. Note that, by default, the CL reader will convert all Symbol names to uppercase at the time the Symbol is read into the system. Note also that Symbols, if evaluated "as is," (i.e. without the `quote`), will by default behave as *variables*, and CL will attempt to retrieve an associated *value*. More on this later.

Here is another example of using `quote` to return a literal expression:

```
(quote (+ 1 2))
```

Unlike evaluating a List expression "as is," this will return the literal List `(+ 1 2)`. This List may now be manipulated as a normal List data structure.

Common Lisp defines the abbreviation `'` as a shorthand way to "wrap" an expression in a call to `quote`. So the previous examples could equivalently be written as:

```
'a

'(+ 1 2)
```

3.1.4 Fundamental CL Data Types

Common Lisp natively supports many data types common to other languages, such as Numbers, Strings, and Arrays. Native to CL is also a set of types which you may not have come across in other languages, such as Lists, Symbols, and Hash tables. In this overview we will touch on Numbers, Strings, Symbols, and Lists. Later in the book we will provide more detail on some CL-specific data types.

Regarding data types, CL follows a paradigm called *dynamic* typing. Basically this means that *values* have type, but *variables* do not necessarily have type, and typically variables are not "pre-declared" to be of a particular type.

Numbers

Numbers in CL form a hierarchy of types, which includes *Integers*, *Ratios*, *Floating Point*, and *Complex* Numbers. For many purposes you only need to think of a value as a "Number," without getting any more specific than that. Most arithmetic operations, such as `+`, `-`, `*`, `/`, etc, will automaticaly do any necessary type coercion on their arguments and will return a Number of the appropriate type.

CL supports a full range of floating-point decimal Numbers, as well as true *Ratios*, which means that `1/3` is a true one-third, not `0.333333333` rounded off at some arbitrary precision.

As we have seen, Numbers in CL are a native data type which simply evaluate to themselves when entered at the toplevel or included in an expression.

Strings

Strings are really a specialized kind of *Array*, namely a one-dimensional array (vector) made up of characters. These characters can be letters, Numbers, or punctuation, and in some cases can include characters from international character sets (e.g. Unicode) such as Chinese Hanzi or Japanese Kanji. The String delimiter in CL is the double-quote character (`"`).

As we have seen, Strings in CL are a native data type which simply evaluate to themselves when included in an expression.

Symbols

Symbols are such an important data structure in CL that people sometimes refer to CL as a "symbolic computing" language. Symbols are a type of CL object which provides your program with a built-in mechanism to store and retrieve Values and Functions, as well as being useful in their own right. A Symbol is most often known by its name (actually a String), but in fact there is much more to a Symbol than its name. In addition to the name, Symbols also contain a *Function* slot, a *Value* slot, and an open-ended *Property-list* slot in which you can store an arbitrary number of named properties.

For a named Function such as `+`, the Function-slot of the Symbol `+` contains the actual Function itself. The Value slot of a Symbol can contain any value, allowing the Symbol to act as a global variable, or *Parameter*. And the Property-list, or *Plist* slot, can contain an arbitrary amount of information.

This separation of the Symbol data structure into Function, Value, and Plist slots is one obvious distinction between Common Lisp and most other Lisp dialects. Most other dialects allow only one (1) "thing" to be stored in the Symbol data structure, other than its name (e.g. either a Function or a Value, but not both at the same time). Because Common Lisp does not impose this restriction, it is not necessary to contrive names, for example for your variables, to avoid conflicting with existing "reserved words" in the system. For example, "`list`" is the name of a built-in Function in CL. But you may freely use "`list`" as a variable as well. There is no need to contrive arbitrary abbreviations such as "`lst`."

How Symbols are evaluated depends on where they occur in an expression. As we have seen, if a Symbol appears *first* in a List expression, as with the `+` in `(+ 2 2)`, the Symbol is evaluated for its Function slot. If the first element of an expression is a Symbol which indeed does contain a Function in its Function slot, then any Symbol which appears *anywhere else* (i.e. in the *rest*) of the expression is taken as a variable, and it is evaluated for its global or local value, depending on its *Scope*. More on Variables and Scope later.

As noted in Section 3.1.3, if you want a literal Symbol itself, one way to achieve this is to "quote" the Symbol name:

```
'a
```

Another way is for the Symbol to appear within a quoted List expression:

```
'(a b c)
```

```
'(a (b c) d)
```

Note that the quote (`'`) applies across everything in the List expression, including any sub-expressions.

Lists

Lisp takes its name from its strong support for the List data structure. The List concept is important to CL for more than this reason alone — Lists are important because *all CL programs themselves*

are Lists! Having the List as a native data structure, as well as the form of all programs, means that it is especially straightforward for CL programs to compute and generate other CL programs. Likewise, CL programs can read and manipulate other CL programs in a natural manner. *This cannot be said of most other languages, and is one of the primary distinguishing characteristics of CL.*

Textually, a List is defined as zero or more elements surrounded by parentheses. The elements can be objects of any valid CL data types, such as Numbers, Strings, Symbols, Lists, or other kinds of Objects. As we have seen, you must quote a literal List to evaluate it or CL will assume you are calling a Function.

Now look at the following List:

```
(defun hello () (write-string "Hello, World!"))
```

This List also happens to be a valid CL program (Function definition, in this case). Don't worry about analyzing the Function right now, but do take a few moments to convince yourself that it meets the requirements for a List. What are the types of the elements in this List?

In addition to using the quote (`'`) to produce a literal List, another way to produce a List is to call the Function `list`. The Function `list` takes any number of arguments, and returns a List made up from the result of evaluating each argument (as with all Functions, the arguments to the `list` Function get evaluated, from left to right, before being passed into the Function). For example,

```
(list 'a 'b (+ 2 2))
```

will return the List `(A B 4)`. The two quoted Symbols evaluate to Symbols, and the Function call `(+ 2 2)` evaluates to the Number `4`.

3.1.5 Functions

Functions form the basic building block of CL. Here we will give a brief overview of how to define a Function; later we will go into more detail on what a Function actually is.

A common way to define named Functions in CL is with the macro `defun`, which stands for DEFinition of a FUNction. `Defun` takes as its arguments a Symbol, an Argument List, and a Body:

```
(defun my-first-lisp-function ()
  (list 'hello 'world))
```

Because `defun` is a *Macro*, rather than a Function, it does not follow the hard-and-fast rule that all its arguments are evaluated as expressions — specifically, the Symbol which names the Function does not have to be quoted, nor does the Argument List. These are taken as a literal Symbol and a literal List, respectively.

Once the Function has been defined with `defun`, you can call it just as you would call any other Function, by wrapping it in parentheses together with its arguments:

```
USER(56): (my-first-lisp-fn)
(HELLO WORLD)

USER(57): (defun square(x)
            (* x x))
SQUARE
```

```
USER(58): (square 4)
16
```

Declaring the types of the arguments to a Function is not required.

3.1.6 Global and Local Variables

Global Variables

Global variables in CL are usually also known as *special* variables. They can be established by calling `defparameter` or `defvar` from the *read-eval-print* loop, or from within a file that you compile and load into CL:

```
(defparameter *todays-temp* 90)
(defvar *humidity* 70)
```

The surrounding asterisks on these Symbol names are part of the Symbol names themselves, and have no significance to CL. This is simply a long-standing naming convention for global variables (i.e. parameters), to make them easy to pick out with the human eye.

`Defvar` and `defparameter` differ in one important way: if `defvar` is evaluated with a Symbol which already has a global value, it will *not* overwrite it with a new value. `Defparameter`, on the other hand, *will* overwrite it with a new value:

```
USER(4): *todays-temp*
90

USER(5): *todays-humidity*
70

USER(6): (defparameter *todays-temp* 100)
*TODAYS-TEMP*

USER(7): (defvar *todays-humidity* 50)
*TODAYS-HUMIDITY*

USER(8): *todays-temp*
100

USER(9): *todays-humidity*
70
```

`*todays-humidity*` did not change because we used `defvar`, and it already had a previous value.

The value for any Parameter can always be changed from the toplevel with setq:

```
USER(11): (setq *todays-humidity* 30)
30

USER(12): *todays-humidity*
30
```

Although `setq` will work with new variables, stylistically it should only be used with already-established variables.

During application development, it often makes sense to use `defvar` rather than `defparameter` for variables whose values might change during the running and testing of the program. This way, you will not unintentionally reset the value of a parameter if you compile and load the source file where it is defined.

Local Variables

New local variables can be introduced with the Macro `let`:

```
USER(13): (let ((a 20)
                (b 30))
            (+ a b))
50
```

As seen in the above example, `let` takes an *assignment section* and a *body*. `Let` is a Macro, rather than a Function[1], so it does not follow the hard-and-fast rule that all its arguments are evaluated. Specifically, the assignment section

```
((a 20)
 (b 30))
```

is not evaluated (actually, the Macro has the effect of transforming this into something else before the CL evaluator even sees it). Because this assignment section is not evaluated by the `let` Macro, it does not have to be quoted, like a List which is an argument to a Function would have to be. As you can see, the assignment section is a List of Lists, where each internal List is a pair whose `first` is a Symbol, and whose `second` is a value (which *will* be evaluated). These Symbols (`a` and `b`) are the local variables, and they are assigned to the respective values.

The *Body* consists of any number of expressions which come after the assignment section and before the closing parenthesis of the `let` statement. The expressions in this Body are evaluated normally, and of course any expression can refer to the value of any of the local variables simply by referring directly to its Symbol. If the Body consists of more than one expression, the final return-value of the Body is the return-value of its last expression.

New Local variables in CL are said to have *lexical* scope, which means that they are only accessible in the code which is textually contained within the body of the `let`. The term *lexical* is derived from the fact that the behavior of these variables can be determined simply by *reading* the text of the source code, and is not affected by what happens during the program's execution.

Dynamic scope happens when you basically mix the concept of global parameters and local `let` variables. That is, if you use the name of a previously established parameter inside the assignment section of a `let`, like this:

```
(let ((*todays-humidity* 50))
  (do-something))
```

[1] See Section 3.8.5 for a discussion of how to recognize Macros and Functions

the parameter will have the specified value not only textually within the the Body of the `let`, *but also* in any Functions which may get called from within the body of the let. The global value of the parameter in any other contexts will not be affected. Because this scoping behavior is determined by the runtime "dynamic" execution of the program, we refer to it as Dynamic scope.

Dynamic scope is often used for changing the value of a particular global parameter only within a particular tree of Function calls. Using Dynamic scope, you can accomplish this without affecting other places in the program which may be referring to the parameter. Also, you do not have to remember to have your code "reset" the parameter back to a default global value, since it will automatically "bounce" back to its normal global value.

Dynamic Scoping capability is especially useful in a *multithreaded* CL, i.e., a CL process which can have many (virtually) simultaneous threads of execution. A parameter can take on a dynamically scoped value in one thread, without affecting the value of the parameter in any of the other concurrently running threads.

3.2 The List as a Data Structure

In this section we will present some of the fundamental native CL operators for manipulating Lists as data structures. These include operators for doing things such as:

1. finding the length of a List
2. accessing particular members of a List
3. appending multiple Lists together to make a new List
4. extracting elements from a List to make a new List

3.2.1 Accessing the Elements of a List

Common Lisp defines the accessor Functions `first` through `tenth` as a means of accessing the first ten elements in a List:

```
USER(5): (first '(a b c))
A

USER(6): (second '(a b c))
B

USER(7): (third '(a b c))
C
```

For accessing elements in an arbitrary position in the List, you can use the Function `nth`, which takes an integer and a List as its two arguments:

```
USER(8): (nth 0 '(a b c))
A

USER(9): (nth 1 '(a b c))
B

USER(10): (nth 2 '(a b c))
C
```

```
USER(11): (nth 12 '(a b c d e f g h i j k l m n o p))
M
```

Note that `nth` starts its indexing at *zero* (0), so `(nth 0 ...)` is equivalent to `(first ...)`, and `(nth 1 ...)` is equivalent to `(second ...)`, etc.

3.2.2 The "Rest" of the Story

A very common operation in CL is to perform some operation on the `first` of a List, then perform the same operation on the `rest` of the List, repeating the procedure until the end of the List, i.e. the Empty List, is reached. The Function `rest` is very helpful in such cases:

```
USER(59): (rest '(a b c))
(B C)
```

As you can see from this example, `rest` returns a List consisting of all but the `first` of its argument.

3.2.3 The Empty List

The Symbol `NIL` is defined by Common Lisp to be equivalent to the Empty List, `()`. `NIL` also has the interesting property that its Value is *itself*, which means that it will always evaluate to itself, whether or not it is quoted. So `NIL`, `'NIL`, and `()` all evaluate to the Empty List, whose default printed representation is `NIL`:

```
USER(14): nil
NIL

USER(15): 'nil
NIL

USER(16): ()
NIL
```

The Function `null` can be used to test whether or not something is the Empty List:

```
USER(17): (null '(a b c))
NIL

USER(18): (null nil)
T

USER(19): (null ())
T
```

As you may have deduced from the above examples, the Symbol `T` is CL's default representation for True. As with `NIL`, `T` evaluates to itself.

3.2.4 Are You a List?

To test whether or not a particular object is a List, CL provides the Function `listp`. Like many other Functions which end with the letter "`p`," this Function takes a single argument and checks whether it meets certain criteria (in this case, whether it qualifies as a List). These *predicate* Functions ending in the letter "`p`" will always return either `T` or `NIL`:

```
USER(20): (listp '(pontiac cadillac chevrolet))
T

USER(21): (listp 99)
NIL

USER(22): (listp nil)
T
```

Note that `(listp nil)` returns `T`, since `NIL` is indeed a List (albeit the empty one).

3.2.5 The Conditional `If`

Before continuing with a number of other basic List Functions, we will cover the macro `if`, which allows simple conditionals. `If` takes three arguments, a *test-form*, a *then-form*, and an *else-form*. When an `if` form is evaluated, it first evaluates its Test-form. If the form returns non-`NIL`, it will evaluate the Then-form, else it will evaluate the Else-form. The nesting of multiple `if` expressions is possible, but not advised; later we will cover other constructs which are more appropriate for such cases.

Here are some simple examples using the `if` macro:

```
USER(2): (if (> 3 4)
             "no"
           "yes")
"yes"


USER(3): (if (listp '("Chicago" "Detroit" "Toronto"))
             "it is a list"
           "it ain't a list")
"it is a list"

USER(4): (if (listp 99)
             "it is a list"
           "it ain't a list")
"it ain't a list"
```

3.2.6 Length of a List

Normally you can use the Function `length` to get the number of elements in a List as an Integer:

```
USER(5): (length '(gm ford chrysler volkswagen))
4
```

```
USER(6): (length nil)
0
```

The `length` Function, as with most of Common Lisp, can itself be implemented in CL. Here is a simplified version of an `our-length` Function which illustrates how `length` might be implemented:

```
(defun our-length (list)
  (if (null list)
      0
    (+ (our-length (rest list)) 1)))
```

Note that the Function uses the Symbol `list` to name its argument, which is perfectly valid as we discussed in section 3.1.4.

In English, this Function says basically: "If the List is empty, its length is zero. Otherwise its length is one greater than the length of its `rest`." As with many Functions which operate on Lists, this *recursive* definition is a natural way to express the `length` operation.

3.2.7 Member of a List

The `member` Function will help you to determine whether a particular item is an element of a particular List. Like many similar Functions, `member` uses `eql` to test for equality, which is one of the most basic equality Functions in CL (see Section 3.8.4 for a discussion of equality in CL). `Eql` basically means the two objects must be the same Symbol, Integer, or actual Object (i.e. the same address in memory).

Member takes two arguments: an Item and a List. If the Item is not in the List, it returns `NIL`. Otherwise, it returns the `rest` of the List, starting from the found element:

```
USER(7): (member 'dallas '(boston san-francisco portland))
NIL

USER(8): (member 'san-francisco '(boston san-francisco portland))
(SAN-FRANCISCO PORTLAND)
```

As with `length`, we could define `member` using a Function definition which is *very* close to the English description of what the Function is supposed to do[2]:

```
(defun our-member (elem list)
  (if (null list)
      nil
    (if (eql elem (first list))
       list
      (our-member elem (rest list)))))
```

[2]The use of the "nested" `if` statement in this example, while functionally correct, is a violation of good CL style. Later we will learn how to avoid nested `if` statements.

In English, you might read this Function to say "If the List is empty, return `NIL`. Otherwise, if the desired item is the same as the `first` of the List, return the entire List. Otherwise, do the same thing on the `rest` of the List."

Note that, for the purposes of any kind of logical operators, returning *any* non-`NIL` value is "as good as" returning `T`. So the possible return-values of the `member` Function are as good as returning `T` or `NIL` as far as any logical operators are concerned.

3.2.8 Getting Part of a List

`Subseq` is a common Function to use for returning a portion of a List (or actually any type of *Sequence*). Subseq takes at least two arguments, a List and an Integer indicating the position to *start* from. It also takes an optional third argument, an Integer indicating the position to *stop*. Note that the position indicated by this third argument is *not* included in the returned sub-List:

```
USER(9): (subseq '(a b c d) 1 3)
(B C)

USER(10): (subseq '(a b c d) 1 2)
(B)

USER(11): (subseq '(a b c d) 1)
(B C D)
```

Note also that the optional third argument in effect defaults to the `length` of the List.

3.2.9 Appending Lists

The Function `append` takes any number of Lists, and returns a new List which results from appending them together. Like many CL Functions, `append` does not *side-effect*, that is, it simply returns a new List as a return-value, but does not modify its arguments in any way:

```
USER(6): (setq my-slides '(introduction welcome lists Functions))
(INTRODUCTION WELCOME LISTS FUNCTIONS)

USER(7): (append my-slides '(numbers))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)

USER(8): my-slides
(INTRODUCTION WELCOME LISTS FUNCTIONS)

USER(9): (setq my-slides (append my-slides '(numbers)))
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)

USER(10): my-slides
(INTRODUCTION WELCOME LISTS FUNCTIONS NUMBERS)
```

Note that the simple call to `append` does not affect the variable `my-slides`. If we wish to modify the value of this variable, however, one way to do this is by using `setq` in combination with the call to `append`. Note also the use of `setq` directly at the toplevel with a new variable name. For testing and "hacking" at the Command Prompt, this is acceptable, but in a finished program all such global variables should really be formally declared with `defparameter` or `defvar`.

3.2.10 Adding Elements to a List

To add a single element to the front of a List, you can use the Function `cons`:

```
USER(13): (cons 'a '(b c d))
(A B C D)
```

`Cons` is actually the low-level primitive Function upon which many of the other List-constructing Functions are built. As you may have guessed, "`cons`" stands for "CONStruct," as in "Construct a List." When you read or hear people talking about a CL program doing a lot of "consing," they are referring to the program's behavior of building a lot of List structures, many of which are transitory and will need to be "freed up" by the automatic memory management subsystem, or "garbage collector."

As with `append`, `cons` is *non-destructive*, meaning it does no side-effecting (modification) to its arguments.

3.2.11 Removing Elements from a List

The Function `remove` takes two arguments, any item and a List, and returns a new List with all occurences of the item taken out of it:

```
USER(15): (setq data '(1 2 3 1000 4))
(1 2 3 1000 4)

USER(16): (remove 1000 data)
(1 2 3 4)

USER(17): data
(1 2 3 1000 4)

USER(18): (setq data (remove 1000 data))
(1 2 3 4)

USER(19): data
(1 2 3 4)
```

Like `append` and `cons`, `remove` is non-destructive and so does not modify its arguments. As before, one way to achieve modification with variables is to use `setq`.

3.2.12 Sorting Lists

The Function `sort` will sort any Sequence (including, of course, a List), based on comparing the elements of the Sequence using any applicable comparison Function, or *predicate* Function. Because of efficiency reasons, the designers of CL made the decision to allow `sort` to modify ("recycle" the memory in) its Sequence argument — so you must always "catch" the result of the sorting by doing something explicitly with the return-value:

```
USER(20): (setq data '(1 3 5 7 2 4 6))
(1 3 5 7 2 4 6)
```

```
USER(21): (setq data (sort data #'<))
(1 2 3 4 5 6 7)

USER(22): (setq data (sort data #'>))
(7 6 5 4 3 2 1)
```

Notice that the comparison Function must be a Function which can take two arguments — any representative two elements in the given sequence — and compare them to give a `T` or `NIL` result. The "`#'`" notation is a shorthand way of retrieving the actual Function object associated with a Symbol or expression (in this case, the `<` or `>` Symbol). We will cover more on this in Section 3.4.

In the above examples, we simply reset the value of the variable `data` to the result of the sort, which is a common thing to do. If one wanted to retain the original pre-sorted sequence, a "safe" version of `sort` could be defined as follows:

```
(defun safe-sort (list predicate)
  (let ((new-list (copy-list list)))
    (sort new-list predicate)))
```

3.2.13 Treating a List as a Set

The Functions `union`, `intersection`, and `set-difference` take two Lists and compute the corresponding set operation. Because mathematical sets have no notion of ordering, the order of the results returned by these Functions is purely arbitrary, so you should never depend on the results of these set operations being in any particular order:

```
USER(23): (union '(1 2 3) '(2 3 4))
(1 2 3 4)

USER(25): (intersection '(1 2 3) '(2 3 4))
(3 2)

USER(26): (set-difference '(1 2 3 4) '(2 3))
(4 1)
```

3.2.14 Mapping a Function to a List

If you have one List, and desire another List of the same length, there is a good chance that you can use one of the mapping Functions. `Mapcar` is the most common of such Functions. `Mapcar` takes a Function and one or more Lists, and *maps* the Function across each element of the List, producing a new resulting List. The term *car* comes from the original way in Lisp of referring to the `first` of a List ("Contents of Address Register"). Therefore `mapcar` takes its Function and applies it to the `first` of each successive `rest` of the List:

```
USER(29): (defun twice (num)
            (* num 2))
TWICE
```

```
USER(30): (mapcar #'twice '(1 2 3 4))
(2 4 6 8)
```

"Lambda" (unnamed) Functions are used *very* frequently with `mapcar` and similar mapping Functions. More on this in Section 3.4.3.

3.2.15 Property Lists

Property Lists ("Plists") provide a simple yet powerful way to handle *keyword-value pairs*. A Plist is simply a List, with an even number of elements, where each pair of elements represents a named value. Here is an example of a Plist:

```
(:michigan "Lansing" :illinois "Springfield"
 :pennsylvania "Harrisburg")
```

In this Plist, the *keys* are Keyword Symbols, and the *values* are Strings. The Keys in a Plist are very often Keyword Symbols. Keyword Symbols are Symbols whose names are preceded by a colon (`:`), and which are generally used just for matching the Symbol itself (i.e. typically they are not used for their Symbol-value or Symbol-function).

For accessing members of a Plist, CL provides the Function `getf`, which takes a Plist and a Key:

```
USER(34): (getf '(:michigan "Lansing"
                  :illinois "Springfield"
                  :pennsylvania "Harrisburg")
                :illinois)
"Springfield"
```

3.3 Control of Execution

"Control of Execution" in CL boils down to "Control of Evaluation." Using some standard and common Macros, you control which forms get evaluated, among certain choices.

3.3.1 If

Perhaps the most basic Macro for imposing control is the `if` Macro, which we have already introduced briefly in Section 3.2.5. The `If` Macro takes exactly three arguments, each of which is an expression which may be evaluated. The first is a *test-form*, the second is a *then-form*, and the third is an *else-form* (which may be left out for a default of `NIL`). When CL evaluates the `if` form, it will first evaluate the `test-form`. Depending on whether the return-value of the Test-form is non-`NIL` or `NIL`, either the Then-form or the Else-form will be evaluated.

If you want to group multiple expressions together to occur as part of the Then-form or the Else-form, you must wrap them in some kind of enclosing block. `Progn` is commonly used for this purpose. `Progn` will accept any number of forms as arguments, and will evaluate each of them in turn, finally returning the return-value of the last of them (see Figure 3.1).

```
(if (eql status :all-systems-go)
    (progn (broadcast-countdown) (flash-colored-lights)
           (play-eerie-music)(launch-rocket))
  (progn (broadcast-apology) (shut-down-rocket)))
```

Figure 3.1: Progn used with If

3.3.2 When

In some ways, `when` is similar to `if`, but you should use `when` in cases where you know that the Else-clause is simply `NIL`. This turns out to be a common situation. Another advantage of using `when` in these cases is that there is no need for `progn` to group multiple forms – `when` simply takes a Test-form then any number of "Then-forms:"

```
USER(36): (when (eql database-connection :active)
            (read-record-from-database)
            (chew-on-data-from-database)
            (calculate-final-result))
999

USER(37): (when (> 3 4)
            (princ "I don't think so..."))
NIL
```

3.3.3 Logical Operators

The Logical Operators `and` and `or` evaluate one or more expressions given as arguments, depending on the return-values of the expressions. As we covered in Section 3.2.3, `T` is CL's default representation for "True," `NIL` (equivalent to the Empty List) is CL's default representation for "False," and any non-`NIL` value is "as good as" `T` as far as "Truth" is concerned:

```
USER(38): (and (listp '(chicago detroit boston new-york))
               (listp 3.1415))
NIL

USER(39): (or (listp '(chicago detroit boston new-york))
              (listp 3.1415))
T
```

In fact what is happening here is that `and` evaluates its arguments (expressions) one at a time, from left to right, returning `NIL` as soon as it finds one which returns `NIL` — otherwise it will return the value of its last expression. `Or` evaluates its arguments until it finds one which returns *non*-`NIL`, and returning that *non*-`NIL` return-value if it finds one. Otherwise it will end up returning `NIL`.

`Not` takes a single expression as an argument and negates it — that is, if the expression returns `NIL`, `not` will return `T`, and if the argument returns non-`NIL`, `not` will return `NIL`. Logically, `not` behaves identically with the Function `null` — but semantically, `null` carries the meaning of "testing for an Empty List," while `not` carries the meaning of Logical Negation:

```
USER(45): (not NIL)
T

USER(46): (not (listp 3.1415))
T
```

3.3.4 Cond

Because the use of "nested" `if` statements is not appropriate, CL provides the Macro `cond` to accomodate this. Use `cond` in situations when in other languages you might use a repeated "If...then...else if...else if...else if..." `Cond` takes as arguments any number of *test-expression* forms. Each of these forms consists of a List whose `first` is an expression to be evaluated, and whose `rest` is any number of expressions which will be evaluated if the first form evaluates to non-`NIL`. As soon as a non-`NIL` form is found, its corresponding expressions are evaluated and the entire `cond` is finished (i.e. there is no need for an explicit "break" statement or anything of that nature):

```
USER(49): (let ((n 4))
            (cond ((> n 10) "It's a lot")
                  ((= n 10) "It's kind of a lot")
                  ((< n 10) "It's not a lot")))
"It's not a lot"
```

Because `T` as an expression simply evaluates to itself, a convenient way to specify a "default" or "catch-all" clause in a `cond` is to use `T` as the final conditional expression:

```
USER(50): (let ((n 4))
            (cond ((> n 10) "It's a lot")
                  ((= n 10) "It's kind of a lot")
                  (t "It's not a lot")))
"It's not a lot"
```

3.3.5 Case

If you want to make a decision based on comparing a variable against a known set of constant values, `case` is often more concise than `cond`:

```
USER(51): (let ((color :red))
            (case color
              (:blue "Blue is okay")
              (:red  "Red is actually her favorite color")
              (:green "Are you nuts?")))
"Red is actually her favorite color"
```

Note that `case` uses `eql` to do its matching, so it will only work with constants which are Symbols (including Keyword Symbols), Integers, etc. It will not work with Strings.

CL also provides the macros `ecase` and `ccase`, which work just like `case` but provide some automatic error-handling for you in cases when the given value does not match any of the keys.

The Symbol `otherwise` has special meaning to the `case` Macro – it indicates a "default," or "catch-all" case:

```
USER(52): (let ((color :orange))
            (case color
              (:blue "Blue is okay")
              (:red  "Red is actually her favorite color")
              (:green "Are you nuts?")
              (otherwise "We have never heard of that!")))
"We have never heard of that!"
```

3.3.6 Iteration

In addition to its innate abilities to do recursion and mapping, CL provides several operators for doing traditional iteration ("looping"). These include macros such as `do`, `dolist`, `dotimes`, and the "everything-including-the-kitchen-sink" iterating macro, `loop`. `Dolist` and `dotimes` are two of the most commonly used ones, so we will cover them here:

```
USER(53): (let ((result 0))
            (dolist (elem '(4 5 6 7) result)
              (setq result (+ result elem))))
22
```

As seen above, `dolist` takes an *initialization-list* and a *body*. The Initialization-list consists of an *iterator-variable*, a *List-form*, and an optional *return-value-form* (which defaults to `NIL`). The Body will be evaluated once with the Iterator-variable set to each element in the List which is returned by the List-form. When the List has been exhausted, the return-value-form will be evaluated and this value will be returned from the entire `dolist`. In other words, the number of iterations of a `dolist` is driven by the length of the List returned by its List-form.

`Dotimes` is similar in many ways to `dolist`, but instead of a List-form, it takes a form which should evaluate to an Integer. The Body is evaluated once with the Iterator-variable set to each integer starting from zero (0) up to *one less than* the value of the Integer-form:

```
USER(54): (let ((result 1))
            (dotimes (n 10 result)
              (setq result (+ result n))))
46
```

In situations when you want the program to exit early from an iteration, you can usually accomplish this by calling `return` explicitly.

3.4 Functions as Objects

3.4.1 Named Functions

When working with CL, it is important to understand that a Function is actually a kind of Object, separate from any Symbol or Symbol-name with which it might be associated. You can access the actual Function-object in a Symbol's Function-slot by calling the Function `symbol-function` on a Symbol, like this:

```
(symbol-function '+)
#<Function +>
```

or this:

```
(symbol-function 'list)
#<Function LIST>
```

As you can see, the *printed representation* of a named Function-object contains the Function's Symbol-name enclosed in pointy brackets, preceded by a pound sign.

You can explicitly call a Function-object with the Function *funcall*:

```
USER(10): (funcall (symbol-function '+) 1 2)
3

USER(11): (setq my-function (symbol-function '+))
#<Function +>

USER(12): (funcall my-function 1 2)
3
```

A shorthand way of writing `symbol-function` is to write `#'` before the name of a Function:

```
USER(10): (funcall #'+ 1 2)
3

USER(11): (setq my-function #'+)
#<Function +>

USER(12): (funcall my-function 1 2)
3
```

3.4.2 Functional Arguments

"Functional Arguments" are arguments to Functions which *themselves* are Function-objects. We have seen this already with `mapcar`, `sort`, and `funcall`:

```
USER(13): (defun add-3 (num)
             (+ num 3))
```

```
ADD-3

USER(14): (symbol-function 'add-3)
#<Interpreted Function ADD-3>

USER(15): (mapcar #'add-3 '(2 3 4))
(5 6 7)

USER(17): (funcall #'add-3 2)
5
```

The idea of passing Functions around to other Functions is sometimes referred to as `higher-order functions`. CL provides natural support for this concept.

3.4.3 Anonymous Functions

Sometimes a Function will be used so seldom that it is hardly worth going to the extent of creating Named Function with `defun`. For these cases, CL provides the concept of the *lambda*, or "anonymous" (unnamed) Function. To use a Lambda Function, you place the entire Function definition in the position where you normally would put just the Symbol which names the Function, identified with the special Symbol `lambda`:

```
USER(19): (mapcar #'(lambda(num) (+ num 3))
                  '(2 3 4))
(5 6 7)

USER(20): (sort '((4 "Buffy") (2 "Keiko") (1 "Judy") (3 "Aruna"))
                #'(lambda(x y)
                    (< (first x) (first y))))
((1 "Judy") (2 "Keiko") (3 "Aruna") (4 "Buffy"))
```

3.4.4 Optional Arguments

Optional arguments to a Function must be specified after any required arguments, and are identified with the Symbol `&optional` in the Argument List. Each optional argument can be either a Symbol or a List containing a Symbol and an expression returning a default value. In the case of a Symbol, the default value is taken to be `NIL`:

```
USER(23): (defun greeting-list (&optional (username "Jake"))
             (list 'hello 'there username))
GREETING-LIST

USER(24): (greeting-list)
(HELLO THERE "Jake")

USER(25): (greeting-list "Joe")
(HELLO THERE "Joe")
```

3.4.5 Keyword Arguments

Keyword arguments to a Function are basically *named* optional arguments. They are defined much the same as optional arguments:

```
(defun greeting-list (&key (username "Jake")
                           (greeting "how are you?"))
  (list 'hello 'there username greeting))
```

But when you call a Function with keyword arguments, you "splice in" what amounts to a Plist containing keyword Symbols for the names of the arguments along with their values:

```
USER(27): (greeting-list :greeting "how have you been?")
(HELLO THERE "Jake" "how have you been?")

USER(28): (greeting-list :greeting "how have you been?" :username "Joe")
(HELLO THERE "Joe" "how have you been?")
```

Note that with keyword arguments, you use a normal Symbol (i.e. *without* the preceding colon) in the *definition* of the Function, but a keyword Symbol as the "tag" when *calling* the Function. There is a logical reason behind this, which you will understand soon.

3.5 Input, Output, Streams, and Strings

Input and Output in CL is done through objects called *streams*. A Stream is a source or destination for pieces of data which generally is connected to some physical source or destination, such as a terminal console window, a file on the computer's hard disk, or a web browser. Each thread of execution in CL defines global parameters which represent some standard Streams:

- `*standard-input*` is the default data source.
- `*standard-output*` is the default data destination.
- `*terminal-io*` is bound to the console (Command Prompt) window, and by default this is identical to both `*standard-input*` and `*standard-output*`.

3.5.1 Read

`Read` is the standard mechanism for reading data items into CL. `Read` takes a single optional argument for its Stream, and reads a single data item from that Stream. The default for the Stream is `*standard-input*`. If you simply type `(read)` at the Command Prompt, CL will wait for you to enter some valid Lisp item, and will return this item. `Read` simply reads, it does *not* evaluate, so there is no need to quote the data being read. You can set a variable to the result of a call to read:

```
(setq myvar (read))
```

The Function `read-line` is similar to `read`, but will read characters until it encounters a new line or end-of-file, and will return these characters in the form of a String. `Read-line` is appropriate for reading data from files which are in a "line-based" format rather than formatted as Lisp expressions.

3.5.2 Print and Prin1

`Print` and `prin1` each take exactly one CL object and output its printed representation to a Stream, which defaults to `*standard-output*`. `Print` puts a space and a newline after the item (effectively separating individual items with whitespace), and `Prin1` outputs the item with no extra whitespace:

```
USER(29): (print 'hello)

HELLO
HELLO
```

In the above example, you see the Symbol `HELLO` appearing twice: the first time is the output actually being printed on the console, and the second is the normal return-value of the call to `print` being printed on the console by the *read-eval-print* loop. `Print` and `Prin1` both print their output *readably*, meaning that CL's `read` Function will be able to read the data back in. For example, the double-quotes of a String will be included in the output:

```
USER(30): (print "hello")

"hello"
"hello"
```

3.5.3 Princ

`Princ`, as distinct from `print` and `prin1`, prints its output in more of a `human`-readable format, which means for example that the double-quotes of a String will be stripped upon output:

```
USER(31): (princ "hello")
hello
"hello"
```

in the above example, as before, we see the output twice: once when it is actually printed to the console, and again when the return-value of the call to `princ` is printed by the *read-eval-print* loop.

3.5.4 Format

`Format` is a powerful generalization of `prin1`, `princ`, and other basic printing Functions, and can be used for almost all output. Format takes a Stream, a *control-string*, and optionally any number of additional arguments to fill in placeholders in the Control-string:

```
USER(33): (format t "Hello There ~a" 'bob)
Hello There BOB
NIL
```

In the above example, `t` is used as a shorthand way of specifying `*standard-output*` as the Stream, and `~a` is a Control-string placeholder which processes its argument as if by `princ`. If you specify either an actual Stream or `T` as the second argument to `format`, it will print by side-effect and return

NIL as its return-value. However, if you specify NIL as the second argument, format does not output to any Stream at all by side-effect, but instead *returns* a String containing what *would* have been output to a Stream if one had been provided:

```
USER(34): (format nil "Hello There ~a" 'bob)
"Hello There BOB"
```

In addition to ~a, there is a wealth of other Format Directives for an extensive choice of output, such as outputting items as if by prin1, outputting Lists with a certain character after all but the last item, outputting Numbers in many different formats including Roman Numerals and English words, etc.

3.5.5 Pathnames

In order to name directories and files on a computer filesystem in an operating-system independent manner, CL provides the *pathname* system. CL Pathnames do not contain system-specific Pathname Strings such as the forward-slash of Unix/Linux filenames or the backward-slash of MS-DOS/Windows filenames. A CL Pathname is basically a structure which supports a constructor Function, make-pathname, and several accessor Functions, such as pathname-name, pathname-directory, and pathname-type. The Pathname structure contains six slots:

- Directory
- Name
- Type
- Device
- Version
- Host

On standard Unix filesystems, only the Directory, Name, and Type slots are relevant. On MSDOS/Windows filesystems, the Device slot is also relevant. On the Symbolics platform, which has a more advanced filesystem, the Version and Host slots also have relevance.

The constructor Function, make-pathname, takes keyword arguments corresponding to the six possible slots of the Pathname.

The following is an example of creating a Pathname which would appear on Unix as /tmp/try.lisp:

```
USER(15): (defparameter *my-path*
              (make-pathname :directory (list :absolute "tmp")
                             :name "readme"
                             :type "txt"))
*MY-PATH*

USER(16): *my-path*
#p"/tmp/readme.txt"
```

As seen in the above example, the :directory slot is technically a List, whose first is a keyword Symbol (either :absolute or :relative), and whose rest is the individual directory components represented as Strings.

3.5.6 File Input and Output

Doing file input and output in CL is basically a matter of combining the concepts of Streams and Pathnames. You must first *open* a file, which entails associating a Stream with the file. CL does provide the `open` Function, which directly opens a file and associates a Stream to it. For a number of reasons, however, generally you will use a Macro called `with-open-file` which not only opens the file, but automatically closes it when you are done with it, and performs additional cleanup and error-handling details for you. `With-open-file` takes a *specification-list* and a *body*. The Specification-list consists of a *stream-variable* (a Symbol) and a Pathname, followed by several optional keyword arguments which specify, for example, whether the file is for input, output, or both. Here is an example of opening a file and reading one item from it:

```
(with-open-file (input-stream *my-path*)
  (read input-stream))
```

No extra keyword arguments are required in the Specification-list in this example, since "read-only" mode is the default for opening a file. In order to write to a file, additional keyword arguments must be given:

```
(with-open-file (output-stream *my-path*
                 :direction :output
                 :if-exists :supersede
                 :if-does-not-exist :create)
  (format output-stream  "Hello There"))
```

3.6 Hash Tables, Arrays, Structures, and Classes

CL provides several other built-in data structures to meet a wide variety of needs. Arrays and Hash Tables support the storage of values in a manner similar to Lists and Plists, but with much faster look-up speed for large data sets. Structures provide another Plist-like construct, but more efficient and structured.

Classes extend the idea of Structures to provide a full object-oriented programming paradigm supporting multiple inheritance, and Methods which can dispatch based on any number of specialized arguments ("multimethods"). This collection of features in CL is called the Common Lisp Object System, or CLOS.

3.6.1 Hash Tables

A Hash Table is similar to what in some other languages is known as an Associative Array. It is comparable to a single-dimensional array which supports keys which are any arbitrary CL object, which will match using `eql`, e.g. Symbols or Integers. Hash Tables support virtually *constant-time* lookup of data, which means that the time it takes to look up an item in a Hash Table will remain stable, even as the number of entries in the Hash Table increases.

To work with Hash Tables, CL provides the constructor Function `make-hash-table` and the accessor Function `gethash`. In order to set entries in a Hash Table, use the `setf` operator in conjunction with the `gethash` accessor Function. `Setf` is a generalization of `setq` which can be used with *places* resulting from a Function call, in addition to just with Symbols as with *setq*. Here is an example of creating a Hash Table, putting a value in it, and retrieving the value from it:

```
USER(35):  (setq os-ratings (make-hash-table))
#<EQL hash-table with 0 entries @ #x209cf822>

USER(36): (setf (gethash :windows os-ratings) :no-comment)
:NO-COMMENT

USER(37): (gethash :windows os-ratings)
:NO-COMMENT
T
```

Note that the `gethash` Function returns *two* values. This is the first example we have seen of a Function which returns more than one value. The first return-value is the value corresponding to the found entry in the Hash Table, if it exists. The second return-value is a Boolean value (i.e. `T` or `NIL`), indicating whether the value was actually found in the Hash Table. If the value is not found, the first return-value will be `NIL` and the second return-value will *also* be `NIL`. This second value is necessary to distinguish from cases when the entry *is* found in the Hash Table, but its value just happens to be `NIL`.

There are several ways to access multiple return-values from a Function, e.g. by using `multiple-value-bind` and `multiple-value-list`.

3.6.2 Arrays

Arrays are data structures which can hold single- or multi-dimensional "grids" full of values, which are indexed by one or more Integers. Like Hash Tables, Arrays support very fast access of the data based on the Integer indexes. You create Arrays using the constructor Function `make-array`, and refer to elements of the Array using the accessor Function `aref`. As with Hash Tables, you can set individual items in the Array using `setf`:

```
USER(38): (setq my-array (make-array (list 3 3)))
#2A((NIL NIL NIL) (NIL NIL NIL) (NIL NIL NIL))

USER(39): (setf (aref my-array 0 0) "Dave")
"Dave"

USER(40): (setf (aref my-array 0 1) "John")
"John"

USER(41): (aref my-array 0 0)
"Dave"

USER(42): (aref my-array 0 1)
"John"
```

The `make-array` Function also supports a myriad of optional keyword arguments for initializing the array at the time of creation.

3.6.3 Structures

Structures support creating your own data structures with named slots, similar to, for example, the Pathname data structure which we have seen. Structures are defined with the Macro `defstruct`,

which supports many options for setting up the instance constructor Functions, slots, and accessor Functions of the Structure. For a complete treatment of Structures, we refer you to one of the CL references or to the on-line documentation for `defstruct`.

3.6.4 Classes and Methods

Classes and Methods form the core of the Common Lisp Object System (CLOS), and add full-blown object-oriented capabilities to Common Lisp.

A complete treatment of CLOS is beyond the scope of this booklet, but we will provide a brief overview. *Classes* are basically "blueprints" or "prototypes" for Objects. *Objects*, in this context, are a kind of structure for grouping together both data and computational functionality. *Methods* are very similar to Functions, but they can be *specialized* to apply to particular combinations of Object types.

CLOS is a *generic function* object-oriented system, which means that Methods exist independently of Classes (i.e. Methods do not "belong to" Classes). Methods can take different combinations of Class instance types as arguments, however, and Methods can be specialized based upon the particular combinations of types of their arguments. Simple CL datatypes, such as Strings and Numbers, are also considered as Classes.

Here is an example of defining a Class and a Method, making an instance of the Class, and calling the Method using the instance as the argument:

```
USER(43): (defclass  box () ((length :initform 10)
                             (width :initform 10)
                             (height :initform 10)))
#<STANDARD-CLASS BOX>

USER(44): (defmethod volume ((self box))
            (* (slot-value self 'length)
               (slot-value self 'width)
               (slot-value self 'height)))
#<STANDARD-METHOD VOLUME (BOX)>

USER(45): (setq mybox (make-instance 'box))
#<BOX @ #x209d135a>

USER(46): (volume mybox)
1000
```

As seen in the above example, the Class definition takes at least three arguments: a Symbol identifying the name of the Class, a Mixin-list which names superclasses (none, in this example), and a List of slots for the Class, with default values specified by the `:initform` keyword argument.

The Method definition is very much like a Function definition, but its arguments (single argument in the case of this example) are specialized to a particular Class type.

When we call the Method, we pass an instance of the Class as an argument, using a *normal argument List*. This is unlike Java or C++, where a Method is considered to be "of" a particular Class instance, and only additional arguments are passed in an argument List.

CL is more consistent in this regard.

Because normal CL data types are also considered to be Classes, we can define Methods which specialize on them as well:

```
USER(47): (defmethod add ((value1 string) (value2 string))
            (concatenate 'string value1 value2))
#<STANDARD-METHOD ADD (STRING STRING)>

USER(48): (defmethod add ((value1 number) (value2 number))
            (+ value1 value2))
#<STANDARD-METHOD ADD (NUMBER NUMBER)>

USER(49): (add "hello " "there")
"hello there"

USER(50): (add 3 4)
7
```

3.7 Packages

Typically, Symbols in CL exist within a particular Package. You can think of Packages as "Area Codes" for Symbols. They are namespaces used within CL to help avoid name clashes.

Package names are generally represented by Keyword Symbols (i.e. Symbols whose names are preceded by a colon). ANSI CL has a "flat" Package system, which means that Packages do not "contain" other Packages in a hierarchical fashion, but you can achieve essentially the same effect by "layering" Packages. Actually, hierarchical Packages have been added to Allegro CL as an extension to the ANSI Standard, and may be added to the ANSI standard at some point. However, the real use of hierarchical Packages is to avoid name clashes with Package names themselves. You can do this in any case simply by separating the components of a Package name with a delimiter, for example, a dot: `:com.genworks.books`.

In a given CL session, CL always has a notion of the *current Package* which is stored in the (dynamic) variable `*package*`. It is important always to be aware of what the current Package is, because if you are in a different Package than you think you are, you can get very surprising results. For example, Functions that are defined in `:package-1` will not necessarily be visible in `:package-2`, so if you attempt to run those Functions from `:package-2`, CL will think they are undefined.

Normally the CL Command Prompt prints the name of the current Package. You can move to a different Package, e.g. `foo`, with the toplevel comand `:package :foo`.

Small programs can be developed in the `:user` Package, but in general, a large application should be developed in its own individual Package.

3.7.1 Importing and Exporting Symbols

Packages can `export` Symbols to make them accessible from other Packages, and `import` Symbols to make them appear to be local to the Package.

If a Symbol is Exported from one Package but not Imported into the current Package, it is still valid to refer to it. In order to refer to Symbols in other Packages, qualify the Symbol with the Package name and a single colon: `package-2:foo`.

If a Symbol is not Exported from the outside Package, you can *still* refer to it, but this would represent a style violation, as signified by the fact that you have to use a double-colon: `package-2::bar`.

3.7.2 The Keyword Package

We have seen many occurences of Symbols whose names are preceded by a colon (`:`). These Symbols actually reside within the `:keyword` Package, and the colon is just a shorthand way of writing and printing them. Keyword Symbols are generally used for enumerated values and to name (Keyword)

Function arguments. They are "immune" to Package (i.e. there is no possibility of confusion about which Package they are in), which makes them especially convenient for these purposes.

3.8 Common Stumbling Blocks

This section discusses common errors and stumbling blocks that new CL users often encounter.

3.8.1 Quotes

One common pitfall is not understanding *when to quote* expressions. Quoting a single Symbol simply returns the Symbol, whereas without the quotes, the Symbol is treated as a variable:

```
USER(6): (setq x 1)
1

USER(7): x
1

USER(8): 'x
X
```

Quoting a List returns the List, whereas without the quotes, the List is treated as a Function (or Macro) call:

```
USER(10): '(+ 1 2)
(+ 1 2)

USER(11): (first '(+ 1 2))
+

USER(12): (+ 1 2)
3

USER(13): (first (+ 1 2))
Error: Attempt to take the car of 3 which is not listp.
```

3.8.2 Function Argument Lists

When you *define* a Function with `defun`, the arguments go inside their own List:

```
USER(19): (defun square (num)
            (* num num))
SQUARE
```

But when you *call* the Function, the argument List comes spliced in directly after the Function name:

```
USER(20): (square 4)
16
```

A common mistake is to put the argument List into its own List when calling the Function, like this:

```
USER(21) (square (4))
Error: Funcall of 4 which is a non-function.
```

If you are used to programming in Perl or Java, you will likely do this occasionally while you are getting used to CL syntax.

3.8.3 Symbols vs. Strings

Another point of confusion is the difference between Symbols and Strings. The Symbol `AAA` and the String `"AAA"` are treated in completely different ways by CL. Symbols reside in Packages; Strings do not. Moreover, all references to a given Symbol in a given Package refer to the same actual address in memory — a given Symbol in a given Package is only allocated once. In contrast, CL might allocate new memory whenever the user defines a String, so multiple copies of the same String of characters could occur multiple times in memory. Consider this example:

```
USER(17): (setq a1 'aaa)
AAA

USER(18): (setq a2 'aaa)
AAA

USER(19): (eql a1 a2)
T
```

EQL compares actual Pointers or Integers – `A1` and `A2` both point to the same Symbol in the same part of memory.

```
USER(20): (setq b1 "bbb")
"bbb"

USER(21): (setq b2 "bbb")
"bbb"

USER(22): (eql b1 b2)
NIL

USER(23): (string-equal b1 b2)
T
```

Again, `EQL` compares pointers, but here `B1` and `B2` point to different memory addresses, although those addresses happen to contain the same String. Therefore the comparison by `string-equal`, which compares Strings character-by-character, rather than the pointers to those Strings, returns `T`.

Another difference between Symbols and Strings is that CL provides a number of operations for manipulating Strings that do not work for Symbols:

```
USER(25): (setq c "jane dates only lisp programmers")
"jane dates only lisp programmers"

USER(26): (char c 3)
\#e

USER(27): (char 'xyz 1)
Error: `XYZ' is not of the expected type `STRING'

USER(29): (subseq c 0 4)
"jane"

USER(30): (subseq c (position \#space c))
" dates only lisp programmers"

USER(31): (subseq c 11 (length c))
"only lisp programmers"
```

3.8.4 Equality

CL has several different predicates for testing equality. This is demanded because of the many different object types in CL, which dictate different notions of equality.

A simple rule of thumb is:

- Use `eql` to compare Symbols, Pointers (such as Pointers to Lists), and Integers
- Use `=` to compare most numeric values
- Use `string-equal` for case-insensitive String comparison
- Use `string=` for case-sensitive String comparison
- Use `equal` to compare other types of objects (such as Lists and single characters)
- For comparing floating-point Numbers which may differ by a very small margin but should still be considered equal, the safest technique is to take the absolute value of their difference, and compare this with an "epsilon" value within a tolerance of zero. In Figure 3.2 is a simple Function which does this, which assumes that a global parameter is defined, `*zero-epsilon*`, as a numeric value very close to zero:

```
(defun nearly-equal? (num1 num2
                      &optional
                      (tolerance *zero-epsilon*))
  (< (abs (- num1 num2)) tolerance))
```

Figure 3.2: Function for Floating-point Comparison

Examples:

```
USER(33): (setq x :foo)
:FOO

USER(34): (eql x :foo)
T

USER(35): (= 100 (* 20 5))
T

USER(36): (string-equal "xyz" "XYZ")
T

USER(37): (setq abc '(a b c))
(A B C)

USER(38): (equal abc (cons 'a '(b c)))
T

USER(39): (equal \#a (char "abc" 0))
T

USER(40): (eql abc (cons 'a '(b c)))
NIL
```

The last expression returns NIL because, although the two Lists have the same contents, they reside in different places in memory, and therefore the pointers to those Lists are different.

Note that several of the Functions which by default use `eql` for matching will take an optional keyword argument `test` which can override this default:

```
USER(54): (member "blue" '("red" "blue" "green" "yellow")
                  :test #'string-equal)
("blue" "green" "yellow")
```

Recall from Section 3.2.7 that `member` returns the `rest` of the List starting at the found element.

3.8.5 Distinguishing Macros from Functions

Since Macros[3] and Functions both occur as the `first` element in an expression, at first glance it may appear difficult to distinguish them. In practice, this rarely becomes an issue. For beginners in the language, the best approach is to make a default assumption that something is a Function unless you recognize it as a Macro. Most Macros will be readily identifiable because one of the following will hold:

- It is a familiar core part of CL, such as `if`, `cond`, `let`, `setq`, `setf`, etc.
- Its name begins with "`def`," such as with `defun`, `defparameter`, etc.

[3] *Special Operators* are one other category of operator in CL, which internally are implemented differently from Macros, but which for our purposes we can consider like Macros.

- Its name begins with "`with-`," as in `with-open-file`, `with-output-to-string`, etc.
- Its name begins with "`do`," for example `dolist` and `dotimes`.

Beyond these general rules of thumb, if you find yourself in doubt about whether something is a Function or a Macro, you can:

1. Check with on-line reference documentation or in a CL reference book (see the Bibliography in Appendix A)
2. Use the `symbol-function` Function to ask your CL session, like this:

```
USER(24): (symbol-function 'list)
#<Function LIST>

USER(25): (symbol-function 'with-open-file)
#<macro WITH-OPEN-FILE @ #x2000b07a>
```

3.8.6 Operations that cons

Because CL programs do not have to free and allocate memory explicitly, they can be written much more quickly. As an application evolves, some operations which "cons" can be replaced by operations which "recycle" memory, reducing the amount of work which the garbage collector (CL's automatic memory management subsystem) has to do.

In order to write programs which conserve on garbage collecting, one technique is simply to avoid the unnecessary use of operations which cons. This ability will come with experience.

Another technique is to use `destructive` operations, which are best used only after one has gained more experience working with CL. Destructive operations are for the most part beyond the scope of this booklet.

Chapter 4

CL as a CASE tool

In the following two chapters, we will illustrate a rudimentary web-based Personal Accounting Application, complete with SQL database table definitions and User Interface. We will implement this application in less than 100 lines of code, using CL and a simple CASE-like Macro extension language embedded in CL.

4.1 Executable Object Model

First, we will describe the Macro language, which allows us to define Object Classes, their Methods, and their relationships in a simplified, concise manner.

This language captures many of the central concepts of a general-purpose "Object modeling" language such as the Unified Modeling Language (UML), with the crucial difference being that the Object model is *directly executable as an application*.

In the next chapter, we will layer an HTML-based web user interface on top of the Object model, making use of Franz Inc.'s "AllegroServe" (CL-based web server) together with their "HTMLgen" (HTML generation system).

The language described in this chapter allows the developer to work using an Object-oriented approach called the *message-passing* paradigm, which is somewhat simplified (and limited) compared to what pure CLOS can do. However, for some types of problems, working in the Message-passing style is sufficient and appropriate, and can simplify the programming process. Remember, when we use Macro extensions on top of CL, we are working in a *superset* of CL, so we do not give up any functionality — we always have the option of dropping back into pure CLOS or CL.

In the Message-passing paradigm of Object-oriented programming, Methods "belong" to Classes — that is, the Methods of a Class can be defined right along with the Class definition, as in Java or C++. This is not possible in pure CLOS, since a CLOS Method may be associated with any arbitrary number of Classes.

Methods which "belong" to particular Classes are also known as *messages*, thus the term "Message-passing."

4.2 Knowledge Base

In addition to implementing a Message-passing style, the Macro extension we illustrate in this chapter also implements some *knowledge base* features, which mainly consist of *caching*, *dependency tracking*, and a *non-procedural* style of programming.

Caching means that the return-value of a call to a Method (Message) is automatically "memoized" by the system, so that if the Method is called again during the same running session, the system can *directly* return the value, and does not have to do the actual computation again.

Dependency Tracking is a necessary complement to Caching — this feature allows the system to detect if a cached value *depends on* another value somewhere else in the system. This way, if the "depended-upon" value is modified ("*bashed*," in KB parlance), then the cached value is known to be *stale* and must be recomputed (and re-cached) the next time it is demanded.

Non-procedural refers to a style of programming, also known as *declarative* programming, in which the programmer need not be concerned with the order of evaluation, i.e. there is no explicit "begin" and "end" to a program. There is just a high-level Object model which specifies *what* the solution to a problem is, not procedurally *how* the problem is to be solved. The language handles the details of exactly which Methods to call, and when.

This general concept of a Knowledge Base has seen a multitude of implementations, mainly in CL.

An example of a successful commercial implementation is the ICAD System from Knowledge Technologies International, whose Allegro CL-based language is known as the ICAD Design Language, or ICAD/IDL[1]

ICAD/IDL is specifically geared towards mechanical engineering and geometric work, and so it has very deep and broad support for geometric Objects (wireframe, surfaces, and solids), a graphical user interface with geometric display capability, as well as integrations with various Computer-Aided Design (CAD) systems, built on top of the basic KB Object description language.

An example of an open-source KB language implementation is the General-purpose Declarative Language, or GDL, which is upwardly compatible[2] with ICAD/IDL and is implemented in a thin layer of CL/CLOS.

The GDL system accepts a subset of ICAD/IDL syntax, but without any of the geometry-related capabilities or graphical user interface. It is intended as a teaching tool and as a substrate for implementing certain kinds of Object-oriented applications in Common Lisp.

ICAD/IDL is available from Knowledge Technologies International at

`http://www.ktiworld.com`

as part of their Knowledge Based Organization (KBO) suite. GDL is available in unsupported form through Sourceforge at

`http://www.sourceforge.net/projects/gdl/`

and in supported package form from Genworks International at

`http://www.genworks.com`

The Personal Accounting example in this and the next chapter should function identically with either ICAD/IDL or GDL.

4.3 IDL and GDL Syntax

4.3.1 Defpart

`Defpart` is the basic Macro for defining Classes in IDL and GDL. The original meaning of "part" refers to physical geometric parts, but here the meaning is extended to include any kind of Functional

[1] ICAD is not to be confused with AutoCAD, a completely different product from a different company. And ICAD/IDL is not to be confused with CORBA's Interface Definition Language, which is also known as "IDL."

[2] *Upward compatibility*, in this context, means that any application written with GDL is capable of being compiled and run in ICAD/IDL, but the converse is not necessarily true.

Object[3]. A part definition (Defpart) maps directly into a Lisp Class definition.

The `defpart` Macro takes three basic arguments:

- a *name*, which is a Symbol.
- a *mixin-list*, which is a List of Symbols naming other `Defparts` from which the current part will inherit characteristics.
- a *specification-plist*, which is spliced in (i.e. doesn't have its own surrounding parentheses) after the Mixin-list, and describes the Object model by specifying Methods, contained Objects (*aggregates*), etc. The Specification-plist typically makes up the bulk of the Defpart.

Here are descriptions of the most common keywords making up the Specification-plist:

Inputs specify information to be passed into the Object instance when it is created.

Attributes are really cached Methods, with expressions to compute and return a value.

Parts specify other Instances to be "contained" within this instance.

Methods are (uncached) Methods "of" the Defpart, and can also take other non-specialized arguments, just like a normal Function.

Figure 4.1 shows a simple example, which contains two Inputs, `:first-name` and `:last-name`, and a single Attribute, `:greeting`. As you can see, a `defpart` is analogous in some ways to a `defun`,

```
(defpart hello ()
  :inputs (:first-name :last-name)
  :attributes
  (:greeting (format nil "Hello, ~a ~a!!" (the :first-name) (the :last-name))))
```

Figure 4.1: Example of Simple Defpart

where the Inputs are like arguments to the Function, and the Attributes are like return-values. But seen another way, each Attribute in a `defpart` is like a Function in its own right.

The referencing Macro "`the`" *shadows* CL's "`the`" (which is a seldom-used type declaration operator). "`The`" in IDL and GDL is a Macro which is used to *reference* the value of other Messages within the same Defpart or within contained Defparts. In the above example, we are using "`the`" to refer to the values of the Messages (Inputs) named `:first-name` and `:last-name`.

4.3.2 Making Instances and Sending Messages

Once we have defined a Defpart such as the example above, we can use the constructor Function `make-part` in order to create an *instance* of it. This Function is very similar to the CLOS `make-instance` Function. Here we create an instance of `hello` with specified values for `:first-name` and `:last-name` (the required Inputs), and assign this instance as the value of the Symbol `mypart`:

```
GDL-USER(16): (setq mypart
                (make-part 'hello :first-name "John"
                                  :last-name "Doe"))
#<HELLO @ #x218f39c2>
```

[3] A *Functional Object* in this context refers to an entity which accepts some *Inputs* and computes some *Outputs*, with the Outputs being uniquely determined based on a given set of Inputs.

As you can see, the return value is an Instance of Class `hello`. Now that we have an Instance, we can use the Macro `the-object` to send Messages to this Instance:

```
GDL-USER(17): (the-object mypart :greeting)
"Hello, John Doe!!"
```

`The-object` is similar to `the`, but as its first argument you must give an expression which evaluates to a part Instance. `The`, by contrast, assumes that the part instance is the lexical variable `self`, which is automatically set within the lexical context of a Defpart.

For convenience, you can also set `self` manually at the CL Command Prompt, and use `the` instead of `the-object` for referencing:

```
GDL-USER(18): (setq self
                (make-part 'hello :first-name "John"
                                  :last-name "Doe"))
#<HELLO @ #x218f406a>

GDL-USER(19): (the :greeting)
"Hello, John Doe!!"
```

In actual fact, `(the ...)` simply expands into `(the-object self ...)`.

4.3.3 Parts

The "Parts" keyword specifies a List of "contained" Instances, where each instance is considered to be a "child" part of the current part. Each child Part is of a specified Type, which itself must be defined as a Defpart before the child part can be instantiated.

Inputs to each instance are specified as a Plist of Keywords and Value expressions, spliced in after the part's name and type specification, which must match the Inputs protocol of the Defpart being instantiated. Figure 4.2 shows an example of a Defpart which contains some Parts. In this

```
(defpart city ()
  :attributes
  (:total-water-usage (+ (the :hotel :water-usage)
                         (the :bank :water-usage)))
  :parts
  ((:hotel :type 'hotel
           :size :large)
   (:bank  :type 'bank
           :size :medium)))
```

Figure 4.2: Defpart Containing Child Parts

example, `hotel` and `bank` are presumed to be already (or soon) defined as Defparts themselves, which each answer the `:water-usage` Message. The *reference chains*:

```
(the :hotel :water-usage)
```

and

```
(the :bank :water-usage)
```

provide the mechanism to access Messages within the child part Instances.

These child parts become instantiated *on demand*, meaning that the first time these instances or any of their Messages are referenced, the actual instance will be created *and* cached for future reference.

4.3.4 Quantified Parts

Parts may be *quantified*, to specify, in effect, an Array or List of part Instances. The most common type of Quantification is called `:series` quantification. See Figure 4.3 for an example of a Defpart

```
(defparameter *presidents-data*
    '((:name
       "Carter"
       :term 1976)
      (:name "Reagan"
       :term 1980)
      (:name "Bush"
       :term 1988)
      (:name "Clinton"
       :term 1992)))

(defpart presidents-container ()
  :optional-inputs
  (:data *presidents-data*)
  :parts
  ((:presidents :type 'president
                :quantify (:series (length (the :data)))
                :parameters (the :data))))
```

Figure 4.3: Sample Data and Defpart to Contain U.S. Presidents

which contains a quantified set of Instances representing U.S. presidents. Each member of the quantified set is fed inputs from a List of Plists, which simulates a relational database table (essentially a "List of Rows").

Note the following from this example:

- in order to quantify a Part, the additional Input keyword `:quantify` is added, with a List consisting of the keyword `:series` followed by an expression which must evaluate to a Number.
- The special Parts keyword `:parameters` allows you to pass a List of Plists directly as the Inputs to a quantified set of child parts. This will also work for passing a single Plist into a non-quantified Part.
- The Keyword `:optional-inputs` is used, which is a hybrid of `:attributes` and `:inputs`, allowing a *default expression* as with `:attributes`, but allowing a value to be passed in on instantiation or from the parent, as with `:inputs`. A passed-in value will override the Default Expression.

4.4 Personal Accounting Application

Now that we have a basic overview of the language, let's examine some of the Objects required to implement a rudimentary Personal Accounting Application.

The Application is expected to keep track of Accounts and Transactions, and to compute Balances, Cash Flows, etc. The Accounts will be Objects which answer the following Messages:

- Name
- Description
- Account Number
- Account Type (*income/expense* or *asset/liability*)
- Beginning Balance
- Current Balance

The Transactions represent movements of money from one account to another. Therefore they are Objects which must answer the following Messages:

- Date
- Source Account
- Target Account
- Payee
- Amount

Now let's see how these Objects come together in order to compute the desired results.

4.4.1 Persistent Objects

Since this is largely a database application, we can start by looking at which Messages in our two Defparts might represent columns in a database table:

A basic financial Account should store its Name, Description, Account Number, Account Type, and Beginning Balance. Its Current Balance is computed (derived) information and is not required to be stored.

A Transaction does not contain derived information for our purposes. All its Messages are basically columns in a database table.

So let us start by defining some database Objects. Because database table definition requires some datatype information to be specified in advance, and because we would like to produce more than one Defpart to represent each table, we will use a special definition syntax for database tables. We have prepared the simple Macro `deftable` for this purpose, as seen in Figure 4.4. As you can see, `deftable` accepts a description of an SQL-style database table. If you are familiar with Standard Query Language (SQL), you will recognize the data types in the above definitions, e.g. `varchar2` for character strings.

`Deftable` then processes the description into two Defparts: one representing an instance of the table itself, and another representing the Rows of the table instance.

The "table" Defpart will contain a quantified set of "Row" Instances. The Values in the individual Rows are represented by ordinary Messages supported by the "Row" Instances. The Table Object supports Methods for Inserting and Deleting Rows from the table, and the Row Object supports Methods for Updating the values in each Row. (*Methods*, in this context, are Messages which take arguments in addition to the implicit `self` argument present in all Messages).

As a side-effect, the Deftable Macro also communicates through a database connection Object to a live SQL database, and either

```
(deftable account ()
  :columns
  ((:name (:varchar2 100))
   (:description (:varchar2 500))
   (:acct_number :number)
   (:acct_type (:varchar2 30))
   (:begin_balance :number)))

(deftable transaction ()
  :columns
  ((:from_acct  :number )
   (:to_acct    :number)
   (:trans_date :date)
   (:amount     :number)
   (:payee      (:varchar2 500))))
```

Figure 4.4: SQL Table Definitions

1. Creates the table in SQL if it does not exist
2. Alters the table (at least attempts to alter the table) if it already exists

This second of the possible side-effects represents a convenient way to do *schema evolution* on the live tables in the database.

4.4.2 Adding Computed Messages

Now we can extend the Account definition to compute the current balance. For this purpose, we can take advantage of the fact that the "Row" Object generated by `deftable` will have been defined to inherit from a (initially empty) *custom Class*, or "mixin." If the SQL table is named `ACCOUNT`, the Defpart representing Rows in this table will be named `account-sql-row`, and the Custom Mixin will be named `custom-account-sql-row-mixin`. Therefore, we can redefine the `custom-account-sql-row-mixin` Class as shown in Figure 4.5 . Instances of the `account-sql-row` will now answer the `:current-balance` Message. The Message is being computed as follows:

1. First notice that `:transaction-list` is a required Input. This represents a List of Objects representing all Transactions currently in the database.
2. The `:current-balance` Message starts by initializing a local variable, `result`, to the beginning balance of the Account (remember that the Row Object represents one particular Account in the Database, i.e. Row in the Account table).
3. Next, an iteration is established which executes once for each transaction in the `:transaction-list`, setting the local variable `transaction` to each Object in this List.
4. Each time through the body of the `dolist`, a conditional is evaluated:
   - If the current Account Object is the same as the "source" account of the Transaction (i.e. the `:from_acct`), the amount is *subtracted* from the result.
   - If the current Account Object is the same as the "target" account of the Transaction (i.e. the `:to_acct`), the amount is *added* to the result.

```
(defpart custom-account-sql-row-mixin ()
  :inputs
  (:transaction-list)

  :attributes
  (:current-balance
   (let ((result (the :begin_balance)))
     (dolist (transaction (the :transaction-list) result)
       (cond ((eql (the-object transaction :from_acct) (the :gen_id))
              (setq result
                (- result (the-object transaction :amount))))
             ((eql (the-object transaction :to_acct) (the :gen_id))
              (setq result
                (+ result (the-object transaction :amount)))))))))
```

Figure 4.5: Adding Messages to an Account's Row Object

- If the current Account Object matches neither the "source" nor the "target" of the Transaction, none of the conditional tests will match, and *nothing* will happen for that particular Transaction.

5. Finally, after all transactions have been processed, the `dolist` will return `result`, which will have the correct value.

Note also the following:

- We use the `:gen_id` Message for matching Accounts and Transactions. This is the default Unique Key for the database table, a column automatically generated by the `deftable` Macro.
- We are using the SQL Database just as "dumb" data storage, not relying on it at all for doing filtering, joining, etc. For very large datasets we will start to use the database's indexing and joining capabilities — but for now we can conveniently do much of our work right here in CL.

4.4.3 Building the Object Hierarchy

Now that we have some basic persistent building blocks, let us put together a simple application. At this point the only user interface to the application will be the CL Command Prompt. In the next chapter, we will attach a simple web-based interface.

At this point we need a "container" Object to hold our collections of Accounts and Transactions. Figure 4.6 shows a first version of such an Object definition. This part has one Object for Accounts and one Object for Transactions. It also computes `:transaction-list` as the List of all rows from the Transactions table, and specifies this Message as *descendant*. Descendant-attributes provides a mechanism for ensuring that this Message will be available in *all* descendant parts from the part in which it is defined, as if it had been explicitly passed as an Input into each of these parts. This way, we are ensuring that `:transaction-list` will be available in our Account Row Objects (children of the `:accounts`), since our custom-mixin requires `:transaction-list` as an Input.

Descendant-attributes should be used sparingly, as their overuse can lead to confusing situations (i.e. Messages coming from unexpected places in the Object hierarchy).

Now, we will add two Messages to our container, one to compute Net Worth and one to compute Cash Flow. Net Worth is the sum of Balances of all accounts of type "Asset/Liability," and Cash

```
(defpart personal-accountant ()

  :attributes (:transaction-list (the :transactions :row-list))

  :descendant-attributes (:transaction-list)

  :parts
  ((:accounts :type 'account-sql-table)
   (:transactions :type 'transaction-sql-table)))
```

Figure 4.6: Simple Container Object for Accounts and Transactions

Flow is the negation of the sum of Balances of all accounts of type "Income/Expense" (the sign is reversed so that Income appears positive and Expenses appear negative).

Figure 4.7 shows our `personal-accountant` Defpart with these two Messages added. Each

```
(defpart personal-accountant ()
  :attributes
  (:net-worth (let ((result 0))
                (dolist (account (the :accounts :row-list) result)
                  (if (eql (make-keyword (the-object account :acct_type))
                           :asset/liability)
                      (setq result (+ result
                                      (the-object account :current-balance))))))

   :cash-flow (let ((result 0))
                (dolist (account (the :accounts :row-list) (- result))
                  (if (eql (make-keyword (the-object account :acct_type))
                           :income/expense)
                      (setq result (+ result
                                      (the-object account :current-balance))))))

   :transaction-list (the :transactions :row-list))

  :descendant-attributes (:transaction-list)

  :parts
  ((:accounts :type 'account-sql-table)
   (:transactions :type 'transaction-sql-table)))
```

Figure 4.7: Container Object with Net Worth and Cash Flow Added

Message is computed in basically the same manner: a `dolist` is used to iterate through all accounts, accumulating the balance of each account of the applicable type into the `result`.

Now our application consists of one *root* Object, `personal-accountant`, with two child Objects, one for the Accounts and one for the Transactions. These three Objects answer Messages as follows:

- `personal-accountant` will answer `:net-worth` and `:cash-flow` Messages.
- Each Row Object in `account-sql-table` will answer a `:current-balance` Message.
- The Row Objects in each of `account-sql-table` and `transaction-sql-table` will answer an `:update!` Message (with arguments).
- `account-sql-table` and `transaction-sql-table` will each answer `:insert!` and `:delete!` Messages (with arguments).

4.4.4 Interacting with the Model

At this point we have just a CL Command Prompt interface to the system. For purposes of illustration, we will enter two Rows into the Accounts table and a transaction into the Transactions table, then query the application for some of our supported Messages:

```
GENACC(20): (setq self (make-part 'personal-accountant))
#<PERSONAL-ACCOUNTANT @ #x20f678e2>

GENACC(21): (the :accounts :row-list)
NIL

GENACC(22): (the :accounts (:insert!
                            (list :name "Checking"
                                  :description "ACME National Back Checking"
                                  :acct_number 01
                                  :acct_type  "asset/liability"
                                  :begin_balance 500)))

GENACC(23): (the :accounts (:insert!
                            (list :name "Utilities"
                                  :description "Home Utilities -- electric, etc."
                                  :acct_number 55
                                  :acct_type  "income/expense"
                                  :begin_balance 0)))
```

The two calls to the `:insert!` Method each added a Row to the database, as well as updating the current Objects[4] so that any *dependent* Objects or Messages will see the change immediately:

```
GENACC(24): (the :accounts :row-list)
(#<ACCOUNT-SQL-ROW @ #x20f67954> #<ACCOUNT-SQL-ROW @ #x20f67982>)

GENACC(25): (mapsend (the :accounts :row-list) :index)
(1 2)

GENACC(26): (the :accounts (:rows 1) :current-balance)
500
```

[4]It should be mentioned that we have simplified the above examples somewhat — in ICAD/IDL, any modifications to a model, as with the calls to `:insert!`, require a bit of extra syntax which we have omitted because it does not add to the principal point of the example.

```
GENACC(27): (the :accounts (:rows 2) :current-balance)
0
```

The `mapsend` Function above takes a List of Objects and sends a given Message to each of them, returning a List of the results. The `(:rows 1)` and `(:rows 2)` syntax is a way of referencing particular members of a quantified set.

Now let us add a transaction, and see that it updates the relevant Messages:

```
GENACC(28): (the :transactions (:insert!
                                (list :trans_date "25-aug-00"
                                      :from_acct 1
                                      :to_acct 2
                                      :payee "Detroit Edison"
                                      :amount 75)))

GENACC(29): (the :accounts (:rows 1) :current-balance)
425

GENACC(30): (the :accounts (:rows 2) :current-balance)
75

GENACC(31): (the :net-worth)
425

GENACC(32): (the :cash-flow)
-75

GENACC(33): (the :transactions (:insert!
                                (list :trans_date "25-aug-00"
                                      :from_acct 1
                                      :to_acct 2
                                      :payee "Michcon Gas"
                                      :amount 125)))

GENACC(34): (the :accounts (:rows 1) :current-balance)
300

GENACC(35): (the :accounts (:rows 2) :current-balance)
200

GENACC(36): (the :cash-flow)
-200

GENACC(37): (the :net-worth)
300
```

So our main Messages are working, through a simple command-line interface for now. A well-rounded personal accounting program would, of course, require certain extensions. Here are a few examples:

- computing cash flow for a specified time period
- allowing expense accounts to have "sub-accounts"
- allowing the creation of budget targets for expense accounts

With a robust Object Model, features such as these can be added *and tested* in an incremental manner. We would simply add new Messages and possibly additional Objects to our Object Hierarchy, building steadily toward the desired functionality.

As new "required functionality" becomes apparent (as it invariably does), our flexible and extensible Object Model stands ready to comply.

4.5 Conclusion

In *about 45 lines of code*, we have written a basic runnable account-balancing program, including the creation of the required SQL database tables and associated Methods for transacting with the database.

Done in pure Common Lisp (i.e. without using any extension Macros), this application would have taken slightly more code, but still not an unreasonable amount.

The main objective of the example is to show what really is possible with Common Lisp by *judiciously* selecting and/or developing some strategic extension Macros.

The `Defpart` Macro is but one approach to automating the programming process; many other approaches certainly exist and have yet to be discovered.

Chapter 5

CL as an Internet Application Server

In this chapter we will extend the `personal-accountant` Defpart we developed in the previous chapter. As we left it, the only obvious way to interact with the application was by evaluating Lisp expressions at a Command Prompt. Now we will convert it into a Web application, accessible by a web browser (e.g. Netscape or MS Internet Explorer) on any machine on the network connected to the machine running the CL process.

To accomplish this, we will make use of some readily-available CL-based tools:

AllegroServe is a webserver developed for use with Common Lisp, under open-source licensing. It is guaranteed to work with Allegro Common Lisp. The main purpose of AllegroServe is to serve dynamic pages using an html[1] generator. Franz Inc.'s stated goal in developing this module was to make it easy to publish complex web pages and add a web interface to a Common Lisp application[2].

GWL Implements an approach to building Web applications using a tree of dynamic Common Lisp Objects to represent the tree of pages in a Web application. It runs atop ICAD/IDL or GDL, and CL-HTTP[3] or AllegroServe[4].

5.1 AllegroServe and HTMLgen

5.1.1 CL-based Webservers

Traditional webservers work by transmitting prepared static files from a Server computer over a network to a web browser running on a Client computer. Beyond this, scripts can be run, as separate processes from the webserver itself, through the Common Gateway Interface, or CGI, to enable a web server to generate the contents of a page based on computed information, or in response to a "fillout-form" submitted by the user.

CL webservers like AllegroServe and CL-HTTP take the concept of dynamically generated content to a new level by embedding the actual webserver *in* a running CL process, and each request is handled by running a Function or Method which responds by sending output to a Stream connected to the requesting web browser. In this scenario, each client request is usually handled by a separate thread of execution in the CL process.

[1] *HTML* is "Hypertext Markup Language," a standard language which can be used for defining the content and formatting of a web page.

[2] Referenced from the Allegroserve Home Page at `http://allegroserve.sourceforge.net`

[3] *CL-HTTP* is another CL-based webserver, written by John C. Mallery of MIT, and is available at `http://wilson.ai.mit.edu/cl-http/cl-http.html`

[4] Referenced from the GWL Home Page at `http://gwl.sourceforge.net`

5.1.2 Generating HTML with AllegroServe's htmlgen

AllegroServe includes a special Macro, called simply `html`, which makes the generation of HTML output very convenient. Since HTML is already very similar to Lists in structure, the mapping is straightforward. Here are some examples:

```
NET.HTML.GENERATOR(26): (html "hello")
hello

NET.HTML.GENERATOR(27): (html (:html "hello"))
<html>hello</html>

NET.HTML.GENERATOR(28): (html (:html (:body "hello")))
<html><body>hello</body></html>

NET.HTML.GENERATOR(29): (html (:html (:body
                                      (:table
                                       :newline
                                       (:tr (:td "hello"))))))
<html><body><table>
<tr><td>hello</td></tr></table></body></html>

NET.HTML.GENERATOR(30): (html (:html (:body
                                      ((:table :width "100%")
                                       :newline
                                       (:tr (:td "hello"))))))
<html><body><table WIDTH="100%">
<tr><td>hello</td></tr></table></body></html>


NET.HTML.GENERATOR(31): (setq name "Joe")
"Joe"

NET.HTML.GENERATOR(32): (html (:html (:body
                                      ((:table :width "100%")
                                       :newline
                                       (:tr (:td (:princ name)))))))
<html><body><table WIDTH="100%">
<tr><td>Joe</td></tr></table></body></html>
```

As you can see, the `html` macro allows you to write List expressions which contain keywords corresponding to the usual HTML tags, and it follows a simple set of rules for converting these expressions into actual HTML. Any kind of normal Lisp expressions are also allowed within a call to the `html` Macro (e.g. conditionals, Function calls, iteration, etc.), so you can compute and generate any kind of required HTML.

Working with the `html` Macro is much more convenient than working with actual HTML, not only because you have the full programmatic capabilities of CL, but also because you have the automatic nesting of the List expressions. There is no need to grapple with named closing tags (e.g. `</TABLE>` or `</HTML>`).

For complete details of working with htmlgen, see documentation in the file `htmlgen.html` which is included in the AllegroServe distribution (`http://opensource.franz.com`).

5.2 GWL

5.2.1 Overview

GWL, or Generative Web Language, designed to work with either ICAD/IDL or GDL (covered in the previous chapter), consists mainly of a primitive Defpart called `base-html-sheet`. To use GWL, you first identify those parts in your Object hierarchy which are to be available as Web pages in your application. Those parts should then mix in (i.e. inherit from) this `base-html-sheet` primitive part, and should define a Method called `:write-html-sheet` which generates the actual HTML corresponding to that "page," or Object.

GWL also supports several built-in Messages to facilitate the creation of hyperlinks from one "page" (Object) to another, as well as the ability for the user to interact with the Object hierarchy by submitting values from "fillout-forms." Form submittals have the effect of modifying ("bashing") the Object hierarchy, so that the dependency tracking inherent in the language will automatically update any dependent elements on other "pages" on-demand as the user visits those pages.

For a more detailed description of GWL, please see the documentation which accompanies its distribution.

5.2.2 Examples

Figure 5.1 shows an example of a simple part which mixes in `base-html-sheet`. When this part is

```
(defpart hello (base-html-sheet)
  :optional-inputs
  (:first-name "Ben"
   :last-name  "Franklin")

  :methods
  ((:write-html-sheet
    ()
    (html (:head (:title "Greeting Page"))
          :newline
          (:body "Hello, "
                 (:princ (the :first-name))
                 " "
                 (:princ (the :last-name)))))))
```

Figure 5.1: Simple Specialization of Base-html-sheet

instantiated in a web browser (using a URL like "`make?part=hello`"), the following HTML would be returned:

```
<head><title>Greeting Page</title></head>
<body>Hello, Ben Franklin</body>
```

This simple example consists of a single root-level page. A typical application will consist of a tree of pages, which we can represent with `:parts` contained within the part corresponding to our root-level page. These `:parts` can have their own `:parts`, which have their own `:parts`, and so on, resulting in a "tree," or hierarchy, of arbitrary depth, breadth, and complexity. Figure 5.2 presents an example of our "hello" page with some child parts: Notice that we have added two `:parts` to this

```
(defpart hello (base-html-sheet)
  :optional-inputs
  (:first-name "Ben"
   :last-name  "Franklin")

  :parts
  ((:diary :type 'daily-diary)
   (:finances :type 'personal-accountant))

  :methods
  ((:write-html-sheet
    ()
    (html (:head (:title "Greeting Page"))
          :newline
          (:body "Hello, "
                 (:princ (the :first-name))
                 " "
                 (:princ (the :last-name))
                 :newline
                 (:p (the (:write-child-links))))))))
```

Figure 5.2: Specialization of Base-html-sheet with Child Parts

page, each of which presumably are themselves some kind of specialization of `base-html-sheet`, as well as a call to the Method `:write-child-links`. This is a built-in Method of `base-html-sheet` which automatically creates hyperlinks pointing to each of the child parts in the page.

When instantiated in the browser, this part now produces the following HTML:

```
<head><title>Greeting Page</title></head>
<body>Hello, Ben Franklin
<p><ul>
<li><a HREF="answer?respondant=diary+root(569)">Diary</a></li>
<li><a HREF="answer?respondant=finances+root(569)">Finances</a></li>
</ul></p></body>
```

The hyperlink ("HREF") Strings in the above HTML will be recognized by the response Method in the webserver (generically defined by GWL), and will allow the server to respond by selecting the correct Object hierarchy and individual Object within that hierarchy. The `:write-html-sheet` Method of the appropriate Object Instance will then be used to produce the actual response page.

5.2.3 Separating the "View" from the Object

Sometimes it is helpful to think of a "core" Object, which computes results, accesses databases, etc., separate from various "views" on that Object, such as an HTML view, a plain text view, or perhaps an XML view. One way to achieve this would be to add a new "writer-method" for each desired view to the main Object. But this would clutter up the main Object with Methods which may never be needed.

A better approach, supported by ICAD/IDL and GDL, is to associate "writer-methods" to the core Object through the use of *output companions*. The details of working with Companions are beyond the scope of this chapter, but you should be aware that the technique exists.

Companions allow you to attach Methods to a particular *combination* of a core Defpart and a desired output format, for example, the Defpart *hello* and the output format *HTML*.

This represents a step beyond the basic Message-passing paradigm, because we are now associating Methods with *two* specialized arguments (i.e. the core Defpart *and* the output format), rather than just the *one* specialized argument as in basic Message-passing. Pure CLOS, of course, allows Methods to be specialized on *any* number of arguments — so the use of Companions still only scratches the surface of the full power of CLOS.

5.3 Web-enabling the Personal Accounting Application

Our web-enabled Personal Accounting Application will consist of a single "root-level" web page which will contain the following elements:

- A hyperlink leading to a Form for Inserting, Updating, and Deleting Accounts
- A hyperlink leading to a Form for Inserting, Updating, and Deleting Transactions
- An HTML Table displaying Current Balances for each Account
- An HTML Table displaying current Net Worth and current total Cash Flow

5.3.1 Standard Database Forms

As our first step, let us prepare the root-level web page containing the two hyperlinks for our two "fillout-forms." For this, we will make use of a pre-defined GWL Defpart called `html-sql-table`. This Defpart takes a `:table-name` as its required Input, and will produce a generic data maintenance form for this table, providing basic Insert, Update, and Delete capabilities. When instantiated, the `html-sql-table` Defpart will contain an instance of a Table Object corresponding to the specified `:table-name`, accessible through the Message `:table`. Therefore, we can produce our initial root-level page using just a slight variation on the `personal-accountant` Defpart we prepared in Figure 4.7, as shown in Figure 5.3. Observe the following about this Defpart, as compared with the one in Figure 4.7:

- We have mixed in (i.e. inherited from) the `base-html-sheet` Defpart, to enable this part to be instantiated as an HTML web page.
- In conjunction with mixing in the `base-html-sheet`, we have added a `:write-html-sheet` method in order to emit the actual HTML for the part.
- We have changed the types of the `:accounts` and `:transactions` to be `html-sql-table`, to allow these parts to function as data maintenance forms with hyperlinks leading to them. *Contained* within these "fillout-form" parts will be the identical Table Objects as we had previously, now accessible through the Message `:table`.
- Anywhere in the Defpart where we had been referring simply to `(the :accounts)` or to `(the :transactions)`, which were the Table Objects, we now refer to `(the :accounts :table)` or `(the :transactions :table)`, respectively.

Figure 5.4 shows the top level page, which at this point only contains two hyperlinks. Clicking the "Accounts" hyperlink will display the table of Accounts, as seen in Figure 5.5.

This table of Accounts is generated automatically by the generic `html-sql-table` Defpart we are using. This page supports Inserting, Updating, and Deleting the individual rows in the table. Updating and Deleting is accomplished by clicking form elements or hyperlinks associated with each row, and Inserting is accomplished by clicking an `insert-form` hyperlink at the bottom of the page. Figure 5.6 shows the "fillout-form" which results from clicking this link.

Once a new Account has been added, the table of Accounts looks like Figure 5.7.

Figures 5.8, 5.9, and 5.10 show the parallel set of pages for the Transaction table.

```
(defpart personal-accountant (base-html-sheet)
  :attributes
  (:net-worth (let ((result 0))
                (dolist (account (the :accounts :table :row-list) result)
                  (if (eql (make-keyword (the-object account :acct_type))
                           :asset/liability)
                      (setq result (+ result
                                      (the-object account :current-balance))))))

   :cash-flow (let ((result 0))
                (dolist (account (the :accounts :table :row-list) (- result))
                  (if (eql (make-keyword (the-object account :acct_type))
                           :income/expense)
                      (setq result (+ result
                                      (the-object account :current-balance))))))

   :transaction-list (the :transactions :table :row-list))

  :descendant-attributes (:transaction-list)

  :parts
  ((:accounts :type 'html-sql-table
              :table-name 'account)

   (:transactions :type 'html-sql-table
                  :table-name 'transaction))
  :methods
  ((:write-html-sheet
    ()
    (html (:head (:title "Genworks Personal Accounting")
                 (:body (the (:write-child-links))))))))
```

Figure 5.3: Code for Home Page of Personal Accounting Application

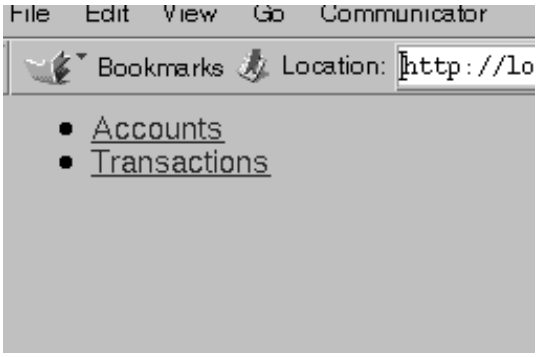

Figure 5.4: Home Page of Personal Accounting Application

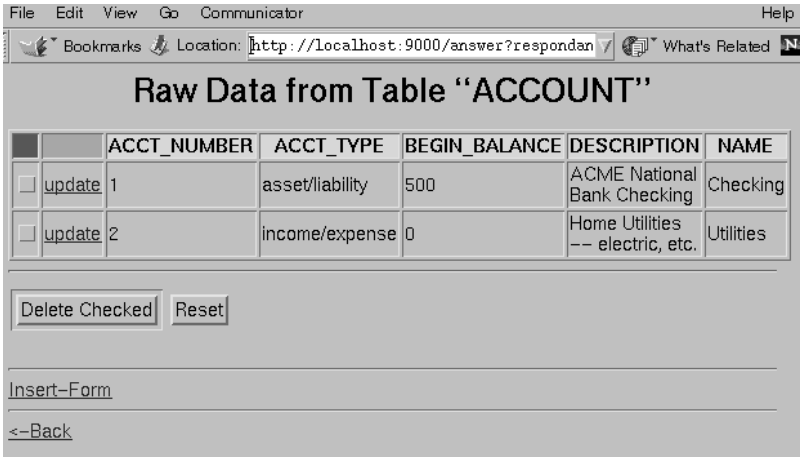

Figure 5.5: Table of Accounts

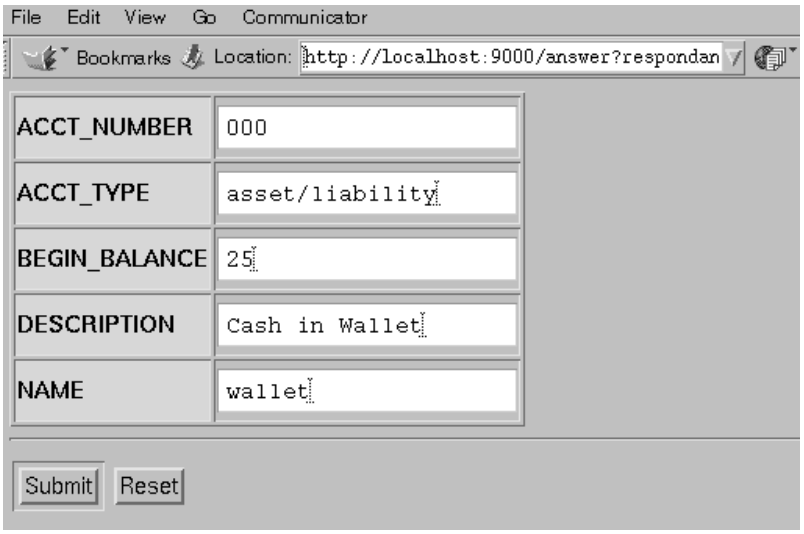

Figure 5.6: Form for Inserting a New Account

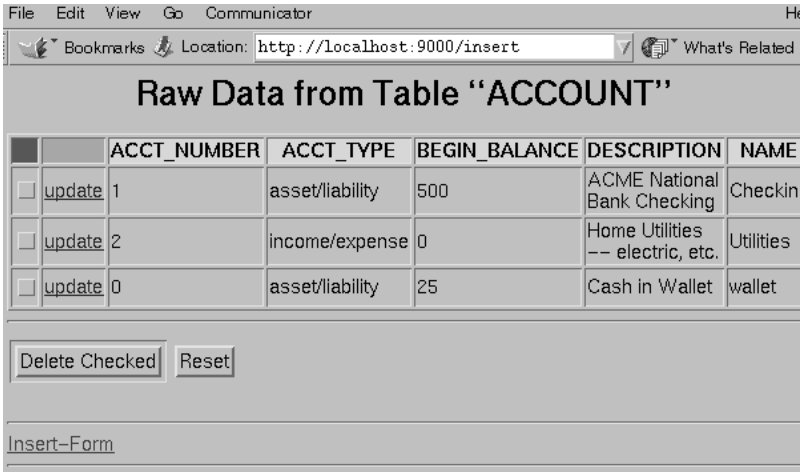

Figure 5.7: Table of Accounts after Inserting a New One

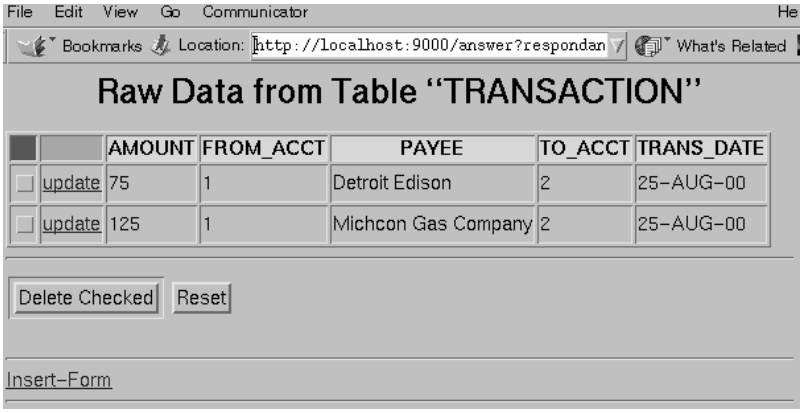

Figure 5.8: Table of Transactions

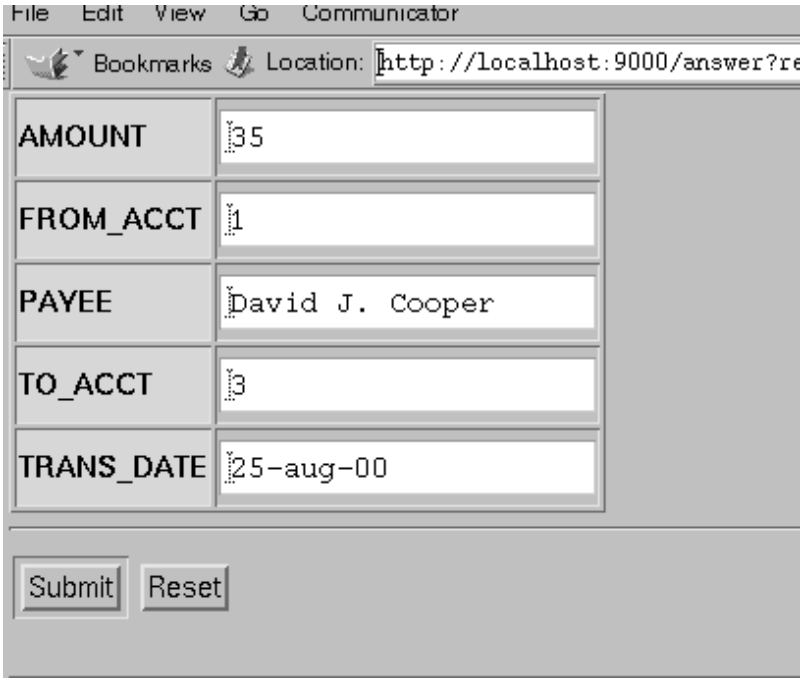

Figure 5.9: Form for Inserting a New Transaction

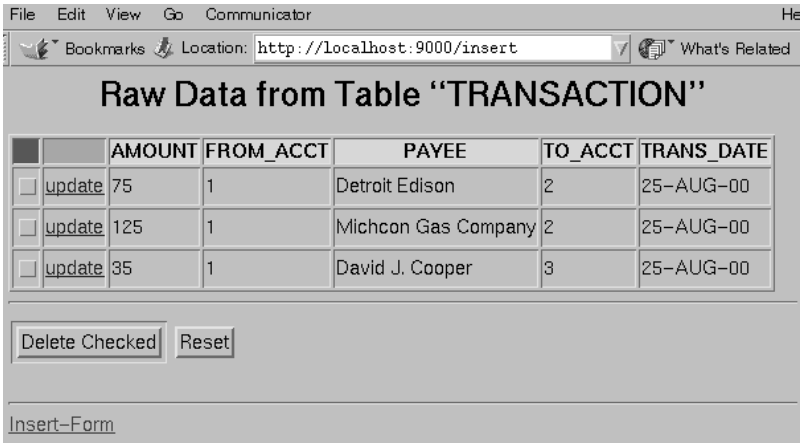

Figure 5.10: Table of Transactions after Inserting a New One

5.3.2 Customizing the Forms

As our application currently stands, inserting and updating Accounts and especially Transactions is not very easy. The user must enter all the data into the forms exactly as it will be stored in the database table.

For example, Transactions are defined as coming "from" one Account "to" another account, where the "from" and "to" must be entered in the form of the `:gen_id` index Numbers. The `:gen_id` is actually intended as a hidden "internal" column of the table, not really for human consumption. A human using this program really wants to select from a List of existing Accounts by *name*. For this purpose, the `html-sql-table` Defpart supports the `:columns-and-names` `:optional-input`, as seen in the snippet of the `personal-accountant` Defpart in Figure 5.11. The `:columns-and-names` is a List of Lists, one for each column in the table. The `first` of each sublist is the Column name, and the `second` is the desired Prompt name. These are followed optionally by the `:choices` Keyword, which can specify that the choices for the field come from a column in a *related* table.

Note the use of the backquote (`) in this snippet. This, together with the comma, is a convenient mechanism for building a literal List, while selectively evaluating certain elements of it (the elements following the commas are evaluated).

Figure 5.12 shows the updated Form, with selection Lists in place of the old type-in fields.

Similarly, the Account table contains the field `:acct_type` which is supposed to be a String with a value of either "income/expense" or "asset/liability." The user will want to pick one of these two choices from a List, not have to type the String exactly. This is accomplished as illustrated in Figure 5.13, which shows just the `:accounts` part specification from `personal-accountant`. This gives a similar result, as seen in Figure 5.14.

Many other customizations are of course possible. But already, with 70 lines of application code, we have prepared a functioning web application which handles the fundamental database table maintenance that we require.

5.3.3 Customizing the Root-level Page with a Report

Looking back at our initial requirements at the beginning of this section, we have yet to place the Current Balances, Net Worth, and Cash Flow on the root-level page. We can accomplish this by extending the `:write-html-sheet` Method of our `personal-accountant` Defpart. First we add an HTML table to display the individual Account Balances (one row per Account). In Figure 5.15 is the relevant portion from the `:write-html-sheet` Method after this is done. Note that the value is negated for accounts of type "Income/Expense," in order to make Income appear positive and Expenses appear negative.

```
:parts
((:accounts :type 'html-sql-table
            :table-name 'account)

 (:transactions :type 'html-sql-table
                :table-name 'transaction
                :columns-and-names
                `(("trans_date" "Date")
                  ("from_acct"
                   "From Account"
                   :choices (:from-table :table-object ,(the :accounts :table)
                                         :key-field :gen_id
                                         :display-field :name))

                  ("to_acct"
                   "To Account"
                   :choices (:from-table :table-object ,(the :accounts :table)
                                         :key-field :gen_id
                                         :display-field :name))

                  ("payee" "Payee")
                  ("amount" "Amount"))))
```

Figure 5.11: Code to Produce Transaction Entry Form with Selection Lists

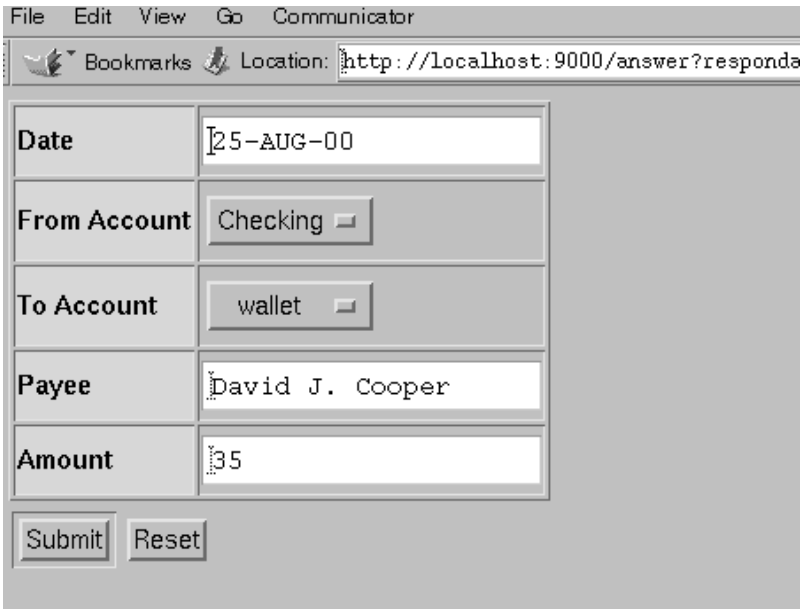

Figure 5.12: Transaction Entry Form with Selection Lists for "From" and "To" Accounts

```
:accounts :type 'html-sql-table
          :table-name 'account
          :columns-and-names
           `(("name" "Name")
             ("description" "Description")
             ("acct_number" "Number")
             ("acct_type"   "Type"
                            :choices (:explicit ("asset/liability"
                                                 "income/expense")))
             ("begin_balance" "Beginning Balance")))
```

Figure 5.13: Code to Produce Account Entry Form with Selection List

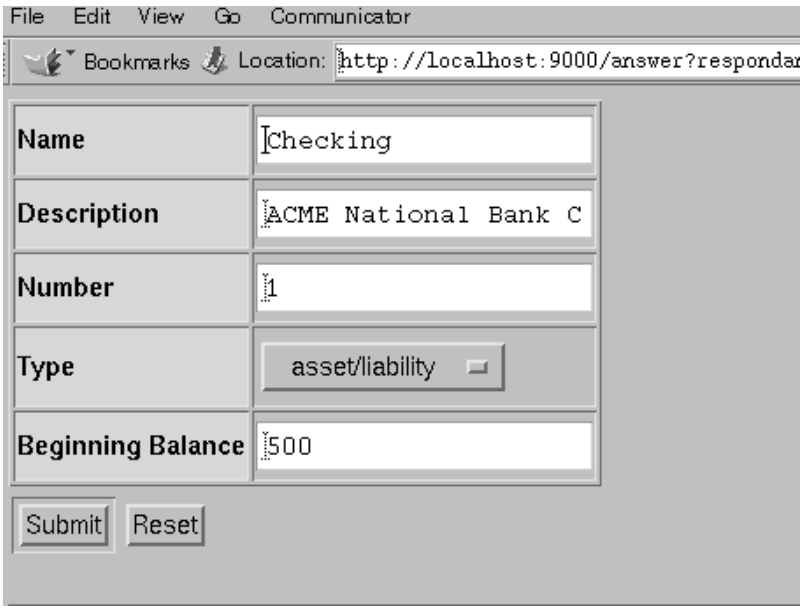

Figure 5.14: Account Entry Form with Selection List for Account Type

```
(:p ((:table :border 1)
     ((:tr :bgcolor "yellow")
      (:th "Account Name")
      (:th "Account Type")
      (:th "Current Balance"))
     (dolist (account (the :accounts :table :row-list))
       (html
        (:tr (:td (:princ (the-object account :name)))
             (:td (:princ (the-object account :acct_type)))
             (let ((value
                    (if (string-equal (the-object account :acct_type)
                                      "income/expense")
                        (- (the-object account :current-balance))
                      (the-object account :current-balance))))
               (html
                ((:td :align "right")
                 (format *html-stream* "~$" value)))))))))
```

Figure 5.15: Displaying Account Balances

Finally, we add a two-row HTML table to display the current values for Net Worth and Cash Flow. Figure 5.16 shows the relevant section of the `:write-html-sheet:` Method. Notice the use

```
(:p ((:table :border 1)
     (:tr ((:td :bgcolor "yellow") "Net Worth")
          ((:td :align "right")
           (format *html-stream* "~$" (the :net-worth))))
     (:tr ((:td :bgcolor "yellow") "Cash Flow")
          ((:td :align "right")
           (format *html-stream* "~$" (the :cash-flow))))))
```

Figure 5.16: Displaying Net Worth and Cash Flow

of the `format` directive `$`, which formats Numbers in a normal currency format (e.g. with two places after the decimal point by default).

The root-level screen now appears as in Figure 5.17.

We now have a usable, web-enabled, SQL-database-driven, reasonably maintainable, highly extensible and customizable Personal Accounting Application, written with some 95 lines of application code.

5.4 Conclusion

This guide has given you a rapid tour of the world of Common Lisp, from the basics of starting and working with CL, through the fundamental parts of the CL language itself, and finally building a working application on top of CL. As you begin your journey of creating your own CL applications, you will experience constant "surprise and delight" as you discover new ways to use its power to accomplish your problem-solving goals.

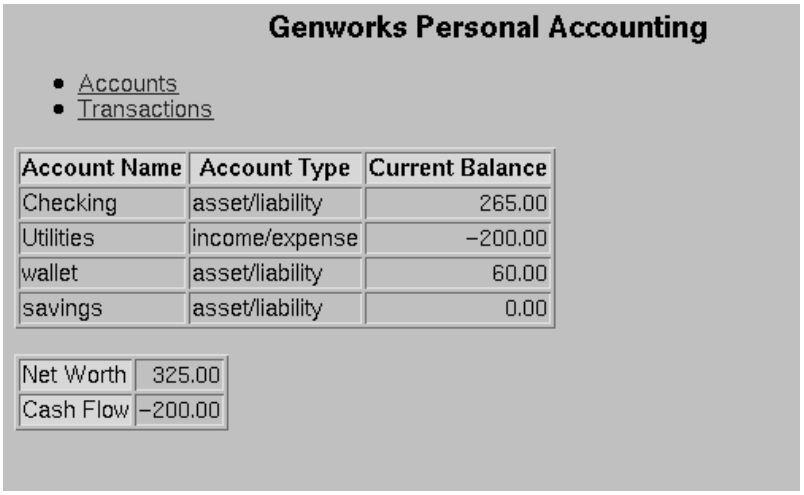

Figure 5.17: Home Page with basic Reports added

Just as CL is an extensible, scalable language, so is the learning process involved. We hope you now have the confidence and motivation to begin immediately putting CL to work for you, solving practical, "every day" programming challenges — and the foundation to move your applications to ever-higher levels.

Appendix A

Bibliography

This Bibliography should also be considered as a Recommended Reading list.

- ANSI Common Lisp, by Paul Graham. Prentice-Hall, Inc, Englewood Cliffs, New Jersey. 1996.
- Common Lisp, the Language, 2nd Edition, by Steele, Guy L., Jr., with Scott E. Fahlman, Richard P. Gabriel, David A. Moon, Daniel L. Weinreb, Daniel G. Bobrow, Linda G. DeMichiel, Sonya E. Keene, Gregor Kiczales, Crispin Perdue, Kent M. Pitman, Richard C. Waters, and Jon L. White. Digital Press, Bedford, MA. 1990.
- ANSI CL Hyperspec, by Kent M. Pitman. Available at

  `http://www.xanalys.com`

- Learning Gnu Emacs, by Eric Raymond. O'Reilly and Associates. 1996.
- Association of Lisp Users website, many other resources and references, at `http://www.alu.org`
- Usenet Newsgroup `comp.lang.lisp`
- Successful Lisp, by David Lamkin, available at

  `http://psg.com/~dlamkins/left/sl/sl.html`

- Franz Website (Commercial CL Vendor), at

  `http://www.franz.com`

- Xanalys Website (Commercial CL Vendor), at

  `http://www.xanalys.com`

- Symbolics Website (Commercial CL Vendor), at

  `http://www.symbolics.com`

- Digitool Website (Macintosh Common Lisp Vendor), at

  `http://www.digitool.com`

- Corman Lisp Website (Commercial CL Vendor), at

  `http://www.corman.net`

- Genworks International Website (services and pointers to other resources) at

  `http://www.genworks.com`

- Knowledge Technologies International Website, at

  `http://www.ktiworld.com`

- Tulane Lisp Tutorial, available at

  `http://www.cs.tulane.edu/www/Villamil/lisp/lisp1.html`

- Texas A&M Lisp Tutorial, available at

  `http://grimpeur.tamu.edu/colin/lp/`

Appendix B

Emacs Customization

B.1 Lisp Mode

When you edit a Lisp file in Emacs, Emacs should automatically put you into Lisp Mode or Common Lisp Mode. You will see this in the status bar near the bottom of the screen.

Lisp Mode gives you many convenient commands for working with List expressions, otherwise known as Symbolic Expressions, or *s-expressions*, or *sexp*'s for short.

For a complete overview of Lisp Mode, issue the command `M-x describe-mode` in Emacs. Table B.1 describes a sampling of the commands available in Lisp Mode[1].

B.2 Making Your Own Keychords

You may wish to automate some commands which do not already have keychords associated with them. The usual way to do this is to invoke some `global-set-key` commands in the `.emacs` file (Emacs' initialization file) in your home directory. Here are some example entries you could put in this file, along with comments describing what they do:

```
;; Make C-x & switch to the *common-lisp* buffer
(global-set-key "\C-x&" '(lambda() (interactive)
                           (switch-to-buffer "*common-lisp*")))

;; Make function key 5 (F5) start or visit an OS shell.
```

[1]Some of these commands are actually available in Fundamental mode as well

Keychord	*Action*	*Emacs Function*
`C-M-space`	Mark sexp starting at point	`mark-sexp`
`M-w`	Copy marked expression	`kill-ring-save`
`C-y`	Paste copied expression	`yank`
`C-M-k`	Kill the sexp starting at point	`kill-sexp`
`C-M-f`	Move point forward by one sexp	`forward-sexp`
`C-M-b`	Move point backward by one sexp	`backward-sexp`
`M-/`	Dynamically expand current word (cycles through choices on repeat)	`dabbrev-expand`

Table B.1: Common Commands of the Emacs-Lisp Interface

```
(global-set-key [f5] '(lambda() (interactive) (shell)))

;; Make sure M-p functions to scroll through previous commands.
(global-set-key "\M-p" 'fi:pop-input)

;; Make sure M-n functions to scroll through next commands.
(global-set-key "\M-n" 'fi:push-input)

;; Enable the hilighting of selected text to show up.
(transient-mark-mode 1)
```

B.3 Keyboard Mapping

If you plan to spend significant time working with your keyboard, consider investing some time in setting it up so you can use it effectively. As a minimum, you should make sure your Control and Meta keys are set up so that you can get at them without moving your hands away from the "home keys" on the keyboard. This means that:

- The Control key should be to the left of the "A" key, assessible with the little finger of your left hand.
- The Meta keys should be on each side of the space bar, accessible by tucking your right or left thumb under the palm of your hand.

If you are working on an XWindow terminal (e.g. a Unix or Linux machine, or a PC running an X Server such as Hummingbird eXceed) you can customize your keyboard with the `xmodmap` Unix command. To use this command, prepare a file called `.Xmodmap` with your desired customizations, then invoke this file with the command

```
xmodmap .Xmodmap
```

Here is an example `.Xmodmap` file for a PC keyboard, which makes the Caps Lock key function as Control, and ensures that both Alt keys will function as Meta:

```
remove Lock = Caps_Lock
keysym Caps_Lock = Control_L
add Control = Control_L

keysym Alt_L = Meta_L Alt_L
```

For PCs running Microsoft Windows, the Windows Control Panel should contain options for customizing the keyboard in a similar manner.

Appendix C

Afterword

C.1 About This Book

This book was typeset using an IBM Thinkpad 1451i running Red Hat Linux and Allegro Common Lisp Professional Edition, using a Lisp-based markup language. For the printed output, the Lisp-based markup language generated LaTeX code and was typeset using LaTeX by Leslie Lamport, which is built atop TeX by Donald Knuth.

C.2 Acknowledgements

I have received inspiration and instruction from many enabling me to write this book. I would like to thank, in no particular order, John McCarthy, Erik Naggum of Naggum Software and comp.lang.lisp, John Foderaro of Franz Inc., Greg Burek, Jeff Moffa, Steve Seippel, and Mark Geoffrey of Ford Motor Company; Stanley Knutson, Garreth Evans, David Howe, Brian Grogan, and the rest of the staff at Knowledge Technologies International; Professor John Laird who taught me CL in college; John Mallery and Paul Graham, who taught me that CL is *the* platform to be using for server applications, James Russell, Andrew Wolven, Dan Fagan, Wuinnie Jimenez, Vassilios Theodoracatos, Craig Lienard, Tom Johnson, and many others whom I am sure I am overlooking here. I would especially like to thank my father, David J. Cooper Sr., Lisa Fettner, Dr. Sheng-Chuan Wu, and Scott Smith for reviewing and commenting on the manuscripts, and Peter Karp from Stanford Research Institute for creating the original outline. "Thank you" to my fiancée Kai Yan, for putting up with my antics while I was writing this (and in general). I would also like to Fritz Kunze of Franz Inc. for telling me to "go write a book."

C.3 About the Author

David J. Cooper has been Chief Engineer with Genworks International since November 1997. His principal activity is working with large manufacturing companies, helping them with their transformation into Knowledge-based Organizations (KBO's). He has led instruction courses in all aspects of KBE, including of course CL, at General Motors, Visteon Automotive, Raytheon Aircraft, Ford Motor Company, Knowledge Technologies International, and others. Prior to his career with Genworks, Mr. Cooper worked with Ford Motor Company, mostly on deployment of CL-based systems in the field of mechanical engineering and manufacturing. He also has software development experience in the CAD and database industries. He received his M.S. degree in Computer Science from the University of Michigan in 1993, and holds a B.S. in Computer Science and a B.A. in German from Michigan as well. His current interests include promoting the use of CL for all kinds of server applications.

Index